AFRL-RI-RS-TR-2009-278
Final Technical Report
December 2009

THE USE OF EMPIRICAL STUDIES IN THE DEVELOPMENT OF HIGH END COMPUTING APPLICATIONS

University of Maryland

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. AO/T810

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-278 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
CHRISTOPHER FLYNN
Work Unit Manager

/s/
EDWARD J. JONES, Deputy Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
DECEMBER 2009	Final	March 2005 – May 2009

4. TITLE AND SUBTITLE

THE USE OF EMPIRICAL STUDIES IN THE DEVELOPMENT OF HIGH END COMPUTING APPLICATIONS

5a. CONTRACT NUMBER
FA8750-05-1-0100

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Victor R. Basili and Marvin V. Zelowitz

5d. PROJECT NUMBER
T810

5e. TASK NUMBER
HE

5f. WORK UNIT NUMBER
CA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Maryland
3112 Lee Building
College Park, MD 20742-5100

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RITB
525 Brooks Road
Rome NY 13441-4505

Defense Advanced Research Projects Agency (DARPA)
3701 North Fairfax Drive
Arlington, VA 22203-1714

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2009-278

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2009-5051 Date Cleared: 4-December-2009

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report provides a description of the research and development activities towards learning much about the development and measurement of productivity in high performance computing environments. Many objectives were accomplished including the development of a methodology for measuring productivity in the parallel programming domain. This methodology was tested over 25 times at 8 universities across the United States and can be used to aid other researchers studying similar environments. The productivity measurement methodology incorporates both development time and performance into a single productivity number. An Experiment Manager tool for collecting data on the development of parallel programs, as well as a suite of tools to aid in the capture and analysis of such data was also developed. Lastly, several large scale development environments were studied in order to better understand the environment used to build large parallel programming applications. That work also included several surveys and interviews with many professional programmers in these environments.

15. SUBJECT TERMS
Productivity, high performance computing (HPC) , parallel programming

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Christopher J. Flynn
U	U	U	UU	92	19b. TELEPHONE NUMBER (*Include area code*) N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENT

Several additional students worked on various aspects of this project including Patrick R. Borek, Daniela Suares Cruzes, and Thiago Escudeiro Craveiro. We'd also like to acknowledge the following faculty for allowing us to conduct experiments in their classes: Alan Edelman [MIT], John Gilbert [UCSB], Mary Hall, Aiichiro Nakano, Jackie Chame [USC], Allan Snavely [UCSD], Alan Sussman, Uzi Vishkin, [UMD], Ed Luke [MSU], Henri Casanova [UH], and Glenn Luecke [ISU]. We would like to acknowledge Chuck Wight, Steve Parker, Anshu Dubey, Edwin van der Wedie, Frank Ham, Gianluca Iaccarino, Dan Meiron, Michael Aivazis, Mark Brandyberry and Robert Fiedler for providing us with the information contained in the ASC case study.. We would also like to thank Robert Voigt for creating the opportunity to conduct these interviews.

SUMMARY

In order to understand how high performance computing (HPC) programs are developed, a series of experiments, using students in graduate level HPC classes and various research centers, were conducted at various locations in the US. In this report, we discuss this research, give some of the early results of those experiments, and describe a web-based Experiment Manager and related tools we are developing that allows us to run studies more easily and consistently at universities and laboratories, allowing us to generate results that more accurately reflect the process of building HPC programs.

This type of empirical research is novel for the HPC community however, so prior to conducting rigorous full fractional factorial experiments with professionals, we ran a pilot studies to debug the experimental methods and techniques. Our first research activity was running a pre-pilot study aimed at understanding the issues involved and debugging our methods. The setting for the pre-pilot study was a graduate level High Performance Computing class taught by Jeffrey K. Hollingsworth at the University of Maryland. The students in this class were taught the concepts associated with HPC, so it was an ideal place to begin evaluating the performance of novice HPC developers. For most of the students in the class, it was their first time developing a HPC software application. Fifteen students took part in the study, which consisted of two assignments. In the first assignment, students were to create a sequential and then a parallel solution using MPI for the "Game of Life" problem. Subjects were graded on, and hence had incentive to focus on, the correct performance of both sequential and parallel versions and the amount of speed-up achieved for the parallel version. In a second assignment, subjects switched to OpenMP and had to devise a parallel version of the SWIM benchmark. Over 4 years, we ran some 25 different studies in a variety of university settings in order to characterize the development process for building HPC programs.

How time is measured can have a pronounced impact on the interpretation of results. We have been conducting empirical studies to characterize how different variables affect programmer effort in the domain of high performance computing. In the context of these studies, we have sought a measure of effort that is both accurate and complete (i.e., captures all programmer activities well). We show how a combination of self-reported and automatic measures of effort data can be used for assessing confidence in results and estimating total effort.

To learn more about the process of developing software to run on HPC systems, we studied five different software projects that develop such codes. These projects make up a group of research centers known as the ASC-Alliance centers. Each center owns a large software project which is focused on addressing a different problem in computational science. These centers are provided with access to large-scale HPC systems located at various supercomputing centers.

To study these projects, we conducted interviews with high-level members of each project. These project members were all involved in project management, software architecture, or software integration. Our goals were to characterize product, project organization, and process, both in terms of using the software and developing the software, and to identify particular challenges faced by the developers. Note that we use the terms "developer", "scientist", and "programmer" interchangeably in this report.

We evolved the Experiment Manager framework to mitigate the complexities of conducting the above mentioned experiments. The framework is an integrated set of tools to support software engineering experiments in HPC classroom (or professional) environments. While aspects of the framework have been studied by others, the integration of all features allows for a uniform environment that has been used in over 25 classroom studies over the past 4 years. The framework supports the following.

1. Minimal disruption of the typical programming process. Study participants solve programming tasks under investigation using their typical work habits, spreading out programming tasks over several days. The only additional activity required is filling out some online forms. Since we do not require them to complete the task in an alien environment or work for a fixed, uninterrupted length of time, we minimize any negative impact on pedagogy or subject overhead.

2. Consistent instruments and artifacts. Use of the framework ensures that the same types of data will be collected and the same types of problems will be solved, which increases confidence in meta-analysis across studies at different universities.

3. Centralized data repository with web interface. The framework provides a simple, consistent interface to the experimental data for experimentalists, subjects, and collaborating professors. This reduces overhead for all stakeholders and ensures that data is consistently collected across studies.

4. Sanitization of sensitive data. The framework provides external researcher with access to the data sets that have been stripped of any information that could identify subjects, to preserve anonymity and comply with the protocols of human subject research as set out by Institutional Review Boards (IRBs) at American universities.

1 INTRODUCTION

The basic goal of the Defense Advanced Research Projects Agency (DARPA) High Productivity Computing System (HPCS) program is to create a new generation of high performance computers operating in the petascale range in number of FLOPS (10^{15} floating point operations per second). The program began in July 2001 when DARPA planned an approximately 10-year program to build a petascale machine [1]. The role of productivity has had a checkered history since that time.

During the initial phase from 2001 until 2004, Five vendors (IBM, SGI, Cray, HP and Sun) were involved in initial design activities. The goal for these new machines, and new languages to execute on them, was a ten-fold improvement in both execution time and development time as measured from the start of the program.

In 2004 at the start of Phase II, the number of vendors was reduced to three (IBM, Cray and Sun) and the concept of productivity was changed. Led by Dr. Jeremy Kepner of Mitre, the University of Maryland became involved and our role in HPCS began in the summer of 2004. Pure counting of Floating Point Operations per Second (FLOPS) was recognized as not a valid criteria for measuring productivity. Few programs achieve sustained execution rates at these high levels – often achieving only 5%-10% of those peak performance rates. In addition, the time and effort to build the programs to run on these machines was not part of the equation. Therefore, the phrase "Time to solution = Development time + Execution time" became the catchword for measuring productivity. The problem to solve, however, was what does "Development time" really mean in the high performance computer arena?

In March of 2005 this initial 3-year grant started when the University of Maryland, aided by the Fraunhofer Center for Experimental Software Engineering and Mississippi State University, began to work with Dr. Kepner and his productivity team in order to understand the mechanisms involved in building high performance codes. Our approach was to develop methods that could be used to establish baseline productivity measures to measure the 2002 productivity as well as the productivity for these new machines when they became operational in the 2010-2012 timeframe.

1.1 Approach toward this research

The HPCS program has goals of "providing a new generation of economically viable high productivity computing systems for national security and for the industrial user community," and initiating "a fundamental reassessment of how we define and measure performance, programmability, portability, robustness and ultimately, productivity in the High Performance Computing (HPC) domain"[1].

In order to reassess the definitions and measures in a scientific domain it is necessary to study the basis and source of those definitions and measures. These sources are usually found in the related literature and various documentations existent in the community. However the large amount of tacit information that is merely in people's minds often remains neglected.

[1] http://www.highproductivity.org

Historically, there has been little interaction between the HPC and the software engineering communities. The Development Time Working Group of the HPCS project was focused on development time issues. The group has both software engineering researchers as well as HPC researchers. The strategy of the working group was to apply empirical methods to study parallel programming issues. We have applied similar methods in the past to researching development time issues in other software domains [2].

Much of the literature in the Software Engineering community concerning programmer productivity was developed with assumptions that do not necessarily hold in the HPC community:

1. In scientific computation insights culled from results of one program version often drives the needs for the next. The software itself is helping to push the frontiers of understanding rather than the software being used to automate well-understood tasks.
2. The requirements often include conformance to sophisticated mathematical models. Indeed, requirements may often take the form of an executable model in a system such as Mathematica, and the implementation involves porting this model to HPC systems.
3. "Usability" in the context of an HPC application development may revolve around optimization to the machine architecture so that computations complete in a reasonable amount of time. The effort and resources involved in such optimization may exceed initial development of the algorithm.

Due to these unique requirements, traditional software engineering approaches for improving productivity may not be directly applicable to the HPC environment.

As a way to understand these differences, we are developing a set of tools and protocols to study programmer productivity in the HPC community. Our initial efforts have been to understand the effort involved and defects made in developing such programs. We also want to develop models of workflows that accurately explain the process that HPC programmers use to build their codes. Issues such as time involved in developing serial and parallel versions of a program, testing and debugging of the code, optimizing the code for a specific parallelization model (e.g., Message Passing Interface (MPI) [3], Open Multi-processing (OpenMP) [4]) and tuning for a specific machine architecture are all topics of study. If we have those models, we can then work on the more crucial problems of what tools and techniques better optimize a programmer's performance to produce quality code more efficiently.

Since 2004 we have been conducting human-subject experiments at various locations across the U.S. in graduate level HPC courses and with interviews with professional programmer at HPC centers (Figure 1). Graduate students in a HPC class are fairly typical of a large class of novice HPC programmers who may have years of experience in their application domain but very little in HPC-style programming. Multiple students are routinely given the same assignment to perform, and we conduct experiments to control for the skills of specific programmers (e.g., experimental meta-analysis) in different environments. Due to the relatively low costs, student studies are an excellent environment to debug protocols that might be later used on practicing HPC programmers.

1.2 Classroom as software engineering lab

The classroom is an appealing environment for conducting software engineering experiments, for several reasons:

- Most researchers are located at universities. Being close to your subjects is often necessary to obtain accurate results.
- Training can be integrated into the course. No extra effort is then required by the subjects since there is the assumption that the training is a valuable academic addition to the classroom syllabus.
- Required tasks can be integrated into the course.
- All subjects are performing identical programming tasks, which is not generally true in industry. This provides an easy source for replicated experiments.

In addition to the results that are obtained directly by these studies, such experiments are also useful for piloting experimental designs and protocols which can later be applied to industry subjects, an approach which has been used successfully elsewhere (e.g., [5] [6] [7]).

While there are threats to validity of such studies by using students as subjects as proxies for professional programmers (e.g., the student environment may not be representative of the ones faced by professional programmers), there are additional complexities that are specific to research in this type of environment. We encountered each of these issues when conducting research on the effects of parallel programming model on effort in high-performance computing [8]:

1. Complexity. Conducting an experiment in a classroom environment is a complex process that requires many different activities (e.g., planning the experimental design, identifying appropriate artifacts and treatments, enrolling students, providing for data collection, checking for process compliance, sanitizing data for privacy, analyzing data). Each such activity identifies multiple points of failure, thus requiring a large effort to organize and run multiple studies. If the study is done at multiple universities in collaboration with other professors, these professors may have no experience in organizing and conducting such experiments.

2. Research vs. pedagogy. When the experiment is integrated into a course, the experimentalist must take care to balance research and pedagogy [9]. Studies must have minimal interference with the course. If the students in one class are divided up into treatment groups and the task is part of an assignment, then care must be taken to ensure that the assignment is of equivalent difficulty across groups. Students who consent to participate must not have any advantage or disadvantage over students who do not consent to participate, which limits additional overhead required by the experiment. In fact, each university's Institutional Review Board (IRB), required in all USA universities performing experiments with human subjects, and insists that participation (or non-participation) must have no effect on the student's grade in the course.

3. Consistent replication across classes. To build empirical knowledge with confidence, researchers replicate studies in different environments. If studies are to be replicated in different classes, then care must be taken to ensure that the artifacts and data collection protocols are consistent. This can be quite challenging because professors have their own style of giving assignments. Common projects across multiple locations often differ in crucial ways making meta-analysis of the combined results impossible [10].

4. Participation overhead for professors. In our experience, many professors are quite willing to integrate software engineering studies into their classroom environment. However, for professors who are unfamiliar with experimental protocols, the more effort required of them to conduct a study, the less likely it will be a success. In addition, collaborating professors who are not empirical researchers may not have the resources or the inclination to monitor the quality of captured data to evaluate process conformance. Therefore, empirical researchers must try to minimize any additional effort required to run an empirical study in the course while ensuring that data is being captured correctly.

The required IRB approval, when attempted for the first time, seems like a formidable task. Help in understanding IRB approval would greatly aid the ability of conducting such research experiments.

5. Participation overhead for students. An advantage of integrating a study into a classroom environment is that the students are already required to perform the assigned task as part of the course, so the additional effort involved in participating in the study is much lower than if subjects were recruited from elsewhere. However, while the additional overhead is low, it is not zero. The motivation to conform to the data collection process is, in general, much lower than the motivation to perform the task, because process conformance cannot be graded. In addition, the study should not subvert the educational goals of the course. Putting the experiment in the context of the course syllabus is never easy.

This can be particularly problematic when trying to collect process data from subjects (e.g. effort, activities, defects), especially for assignments that take several weeks. (e.g., We saw a reduction in process conformance over time when subjects had to fill out effort logs over the course of multiple assignments).

6. Automatic data collection of software process. To reduce subject overhead and increase data accuracy, it is possible to collect data automatically from the programmer's environment. Capturing data at the right level of granularity is difficult. All user-generated events can be captured (keyboard events, mouse events), but this produces an enormous volume of data that may not abstract to useful information. Allowing this raw data to be used can create privacy issues, such as revealing account names, with the ability to then determine how long specific users took to build a product or how many defects they made.

All development activities taking place within a particular development environment (e.g., Eclipse [11]) simplifies the task of data collection, and tools exist to support such cases (e.g. Marmoset [12]). However, in many domains development will involve a wide range of tools and possibly even multiple machines. For example in the domain of high-performance computing, preliminary programs may be compiled on a home PC, final programs are developed on the university multiprocessor and are ultimately run on remote supercomputers at a distant datacenter. Programmers typically use a wide variety of tools, including editors, compilers, build tools, debuggers, profilers, job submission systems, and even web browsers for viewing documentation.

7. Data management. Conducting multiple studies generates an enormous volume of heterogeneous data. Along with automatically collected data and manually reported data, additional data includes versions of the programs, pre- and post-questionnaires, and various quality outcome measures (e.g. grades, code performance, defects). Because of privacy issues, and to conform to IRB regulations, all data must be stored with appropriate access controls, and any exported data must be appropriately sanitized. Managing this data manually is labor-intensive and error-prone, especially when conducting studies at multiple sites.

Limitations of student studies include the relatively short programming assignments possible due to the limited time in a semester and the fact these assignments must be picked for the educational value to the students as well as their investigative value to the research team.

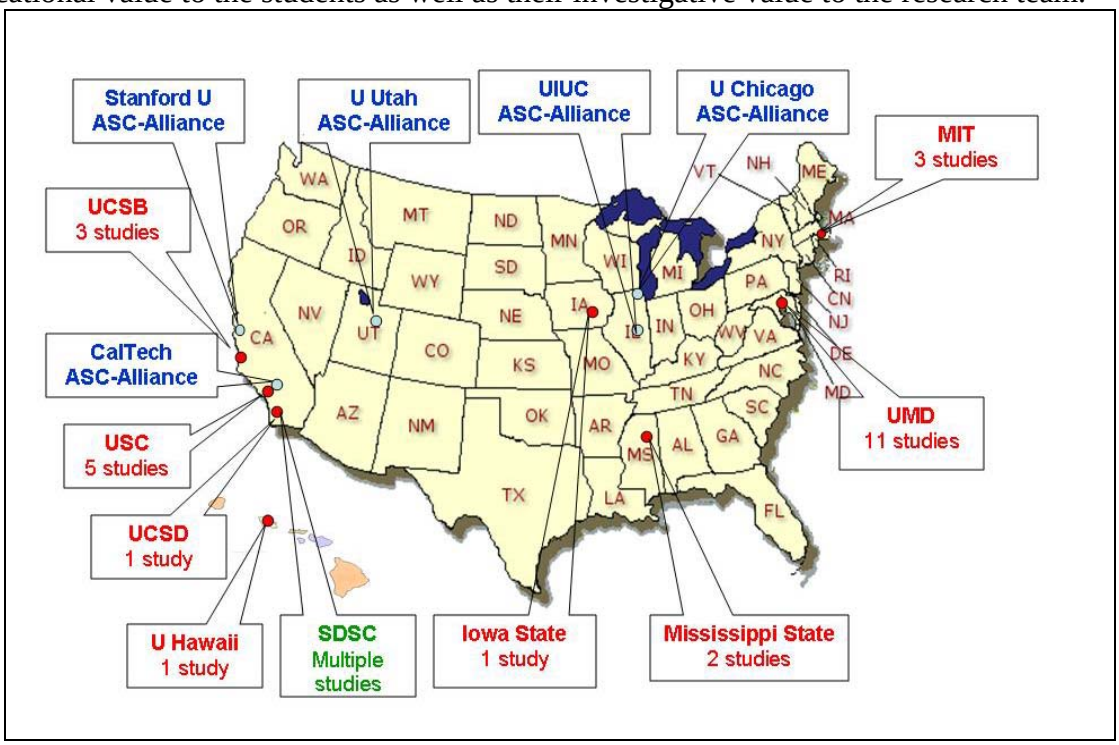

Figure 1: Studies conducted

In this report, we present both the methodology we have developed to investigate programmer productivity issues in the HPC domain (Section 2), some initial results of studying productivity of novice HPC programmers (Section 3).

2 EXPERIMENTAL METHODOLOGY

In each class, we obtained consent from students to be part of our study. There is a requirement at every U.S. institution that studies involving human subjects must be approved by that university's Institutional Review Board (IRB). The nature of the assignments was left to the individual instructors for each class since instructors had individual goals for their courses and the courses themselves had different syllabi. However, based on previous discussions as part of this project, many of the instructors used the same assignments (Table 1), and we have been collecting a database of project descriptions as part of our Experiment Manager website (See Section 3.1). To ensure that the data from the study would not impact students' grades (and a requirement of almost every IRB), our protocol quarantined the data collected in a class from professors and teaching assistants for that class until final grades had been assigned.

Table 1: Sample programming assignments

Embarrassingly parallel:
Buffon-Laplace needle problem, Dense matrix-vector multiply
Nearest neighbor:
Game of life, Sharks & fishes, Grid of resistors, Laplace's equation, Quantum dynamics
All-to-all:
Sparse matrix-vector multiply, Sparse conjugate gradient, Matrix power via prefix
Shared memory:
LU decomposition, Shallow water model, Randomized selection, Breadth-first search
Other:
Sorting

We need to measure the time students spend working on programming assignments with the task that they are working on at that time (e.g. serial coding, parallelization, debugging, tuning). We used three distinct methods: (1) explicit recording by subject in diaries (either paper or web-based); (2) implicit recording by instrumenting the development environment; and (3) sampling by an operating system installed tool (e.g., Hackystat [13]). Each of these approaches has strengths and limitations. But significantly, they all give different answers. After conducting a series of tests using variations on these techniques, we settled on a hybrid approach that combines diaries with an instrumented programming environment that captures a time-stamped record of all compiler invocations (including capture of source code), all programs invoked by the subject as a shell command, and interactions with supported editors. Elsewhere [14], we describe the details of how we gather this information and convert it into a record of programmer effort.

After students completed an assignment, the data was transmitted to the University of Maryland, where it was added to our Experiment Manager database. Looking at the database allows post-project analysis to be conducted to study the various hypotheses we have collected via our folklore collection process.

For example, given workflow data from a set of students, the following hypotheses that are the subjective opinion of many in the HPCS community, collected via surveys at several HPCS meetings, can be tested [15]:

Hypothesis 1: *The average time to fix a defect due to race conditions will be longer in a shared memory program compared to a message-passing program.* To test this hypothesis we can measure the time to fix defects due to race conditions.

Hypothesis 2: *On average, shared memory programs will require less effort than message passing, but the shared memory outliers will be greater than the message passing outliers.* To test this hypothesis we measure the total development time.

Hypothesis 3: *There will be more students who submit incorrect shared memory programs compared to message-passing programs.* To test this hypothesis we can measure the number of students who submit incorrect solutions.

Hypothesis 4: *An MPI implementation will require more code than an OpenMP implementation.* To test this hypothesis we can measure the size of code for each mplementation.

The classroom studies are the first part of a larger series of studies we are conducting (Figure 2). We first run pilot studies with students. We next conduct classroom studies, and then move onto controlled studies with experienced programmers, and finally conduct experiments in situ with development teams. Each of these steps contributes to our testing of hypothesizes by exploiting the unique aspects of each environment (i.e., replicated experiments in classroom studies and multi-person development with in situ teams). We can also compare our results with recent studies of existing HPC codes [16].

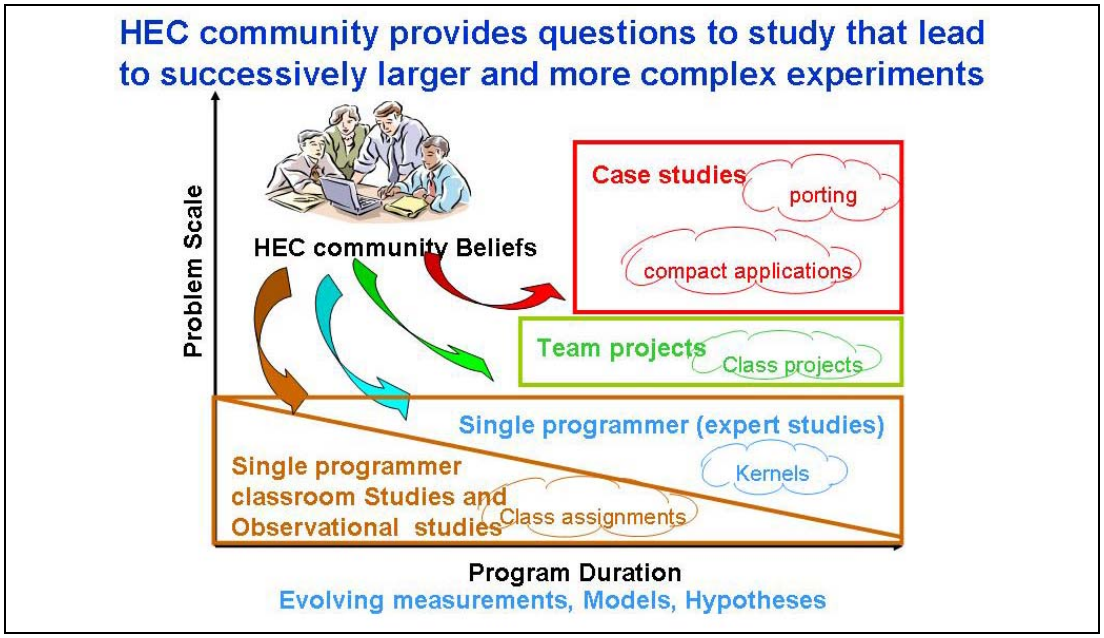

Figure 2: Research plan

2.1 Collection of folklore data

One of the main goals of the development time working group of HPCS project is to leverage HPC community's HPC community's knowledge of development time issues (

Figure 3). In order to do so, we are soliciting expert opinion on issues related to HPC programming by collecting elements of folklore through surveys, generating discussion among experts on these elements of the lore to increase precision of statements and to measure degree of consensus and finally generate testable hypotheses based on the lore that can be evaluated in empirical studies.

Before starting the exploratory experiment of collecting peoples' anecdotal beliefs through surveys, we needed an initial set of such anecdotes to both encourage thinking and also use as examples of what we are interested in.

To gather the folklore in HPC, a member of the study group, who is an HPC professor, conducted an informal scan of several sources including lecture notes used in introductory HPC classes at the University of Maryland as well as scanning the Internet for related keywords (including "HPC tribal lore" and "HPC folklore"). The goal of this process was not to be exhaustive, but instead to gather a sense of the type of information that a beginning HPC programmer might find. This initial list of 10 ideas (the left column of the table in Appendix 1) was recorded and used as the basis for our first survey.

We then asked 7 HPC specialists and professors who regularly teach HPC classes to comment on the initial list. They were asked to give an "agree", "disagree" or "don't know" answer to each candidate, give their comments or change suggestions and add any folklore that they are aware of but is not on the list.

The folklore number 11 in Appendix 1 was added by one of the participants at this stage. Generally the comments revolved around clarifying the domain to which the bit of lore applied. For example was the bit of lore talking about a user programming model such as OpenMP or hardware architecture such as a multi-threaded machine.

In order to clarify the questionable points we scheduled a discussion session among the participants. This discussion resulted in some modifications in the way folklore sentences were phrased. The right column of the table in Appendix 1 is the result of this modification.

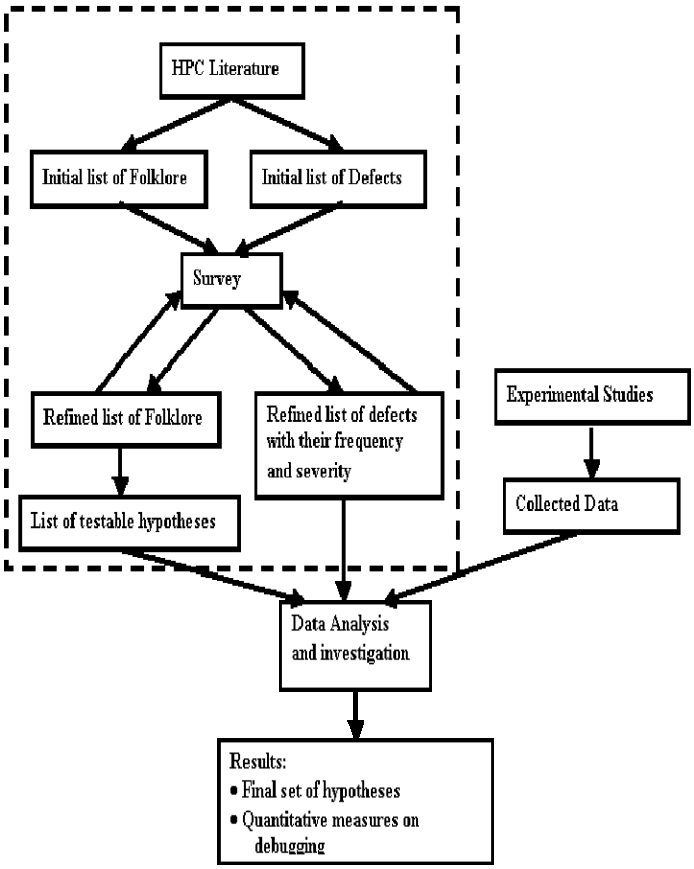

Figure 3:Folklore and defect solicitation process

At some point during the discussion, the participants agreed that "MPI programs don't run well when you use lots of small messages because you get latency-limited". In order to include this in the folklore list, the lore number 12 was added to the list.

At the next step of the study, a survey form was compiled from the current list of 12 folklore and distributed to the participants at the "High Productivity Computing Systems, Productivity Team Meeting" held in January 2005. In order to avoid any bias, some of the randomly selected lore were rephrased to imply the logically inverse sentence. Two sets of survey forms were compiled and distributed randomly.

2.2 Defect studies

As part of our effort to understand development issues, our classroom experiments have moved beyond effort analysis and looked at the impact of defects (e.g. incorrect or excessive synchronization, incorrect data decomposition) on the development process. By understanding how, when, and the kind of defects that appear in HPC codes, tools and techniques can be developed to mitigate these risks to improve the overall workflow. As we have shown [14], automatically determining workflow is not precise, so we worked on a mixture of process activity (e.g., coding, compiling, executing) with source code analysis techniques. The process of defect analysis we built consisted of the following main activities:

- **Analysis**:

1. Analyze successive versions of the developing code looking for patterns of changes represented by successive code versions (e.g., defect discovery, defect repair, addition of new functionality).
2. Record the identified changes.
3. Develop a classification scheme and hypotheses.

For example, a small increase in source code, following a failed execution, following a large code insertion could represent the pattern of the programming adding new functionality followed by a test and then defect correction. Syntactic tools that find specific defects can be used to aid the human-based heuristic search for defects.

- **Verification**:

We then need to analyze these results at various levels. Verification consists of the following steps, among others:

1. If we can somehow obtain the "true" defect sets, we can directly compare our analysis results with them to evaluate the analysis results quantitatively.
2. Multiple analysts can independently analyze the source code and record identified defects.
3. Examine individual instances of defects to check if each defect is correctly captured and documented.
4. Provide defect instances and classify them into one of the given defect types. This can be used to check the consistency of the classification scheme.

[25] explores this defect methodology in greater detail.

3 EXPERIMENTAL RESULTS

An early result needed to validate our process was to verify that students could indeed produce good HPC codes and that we could measure their increased performance. Table 2 is one set of data that shows that students achieved speedups of approximately 3 to 7 on an 8-processor HPC machine. (CxAy means class number x, assignment number y. This coding was used to preserve anonymity of the student population.)

Table 2: Mean, standard deviation, and number of subjects for computing speedup on game of life

Data set	**Programming Model**	**Speedup on 8 processors**
Speedup measured relative to serial version:		
C1A1	MPI	mean 4.74, sd 1.97, n=2
C3A3	MPI	mean 2.8, sd 1.9, n=3
C3A3	OpenMP	mean 6.7, sd 9.1, n=2
Speedup measured relative to parallel version run on 1 processor:		
C0A1	MPI	mean 5.0, sd 2.1, n=13
C1A1	MPI	mean 4.8, sd 2.0, n=3
C3A3	MPI	mean 5.6, sd 2.5, n=5
C3A3	OpenMP	mean 5.7, sd 3.0, n=4

Measuring productivity in the HPC domain is part of understanding HPC workflows; however, what does productivity mean in this domain [19]? The following is one model that we can derive from the fact that the critical component of HPC programs is the speedup achieved by using a multiprocessor HPC machine over a single processor [26]:

$$\text{Speedup} = \frac{\text{Reference Execution Time}}{\text{Parallel Execution Time}}$$

$$\text{Productivity} = \frac{\text{Relative Speedup}}{\text{Relative Effort}} = \frac{\rho}{1/\varepsilon} = \rho \times \varepsilon$$

$$\text{Relative Effort} = \frac{\text{Parallel Effort}}{\text{Reference Effort}}$$

Rho (ρ) represents the relative speedup achieved, while epsilon (ε) represents the effort to produce a serial version of the program. Productivity is defined as the relative speedup of a program using an HPC machine compared to a single processor divided by the relative effort to produce the HPC version of the program divided by the effort to produce a single processor version of the program.

Table 3 shows the results for one group of students programming the Game of life (a simple nearest neighbor cellular automaton problem where the next generation of "life" depends upon

Problem	serial	MPI	OpenMP	Matlab*P	XMT-C	Co-Array Fortran	UPC	Hybrid MPI-OpenMP
Game of life	4	5	2	1		2	2	
SWIM			1					
Buffon-Laplace	2	3	2	3				
Laplace's equation	1	1	1	1				
Sharks & fishes	1	2	2			1		
Grid of resistors	1	1	1	1				
Matrix power via prefix		3	1			1	1	
Sparse conjugate-gradient		2				1	1	
Dense matrix-vector multiply	1	1	1					
Sparse matrix-vector multiply	1	1			2			
Sorting	2	3	1		2			
Quantum dynamics		2						
Molecular dynamics								1
Randomized selection					1			
Breadth-first search					1			
LU decomposition			1					
Shortest path			1					
Search for intelligent puzzles		1						

surrounding cells in a grid and a popular first parallel program for HPC classes) [20]. The data shows that our definition of productivity had a negative correlation compared to both total effort and HPC execution time, and a positive correlation compared to relative speedup. While the sample size is too small for a test of significance, the relationships all indicate that productivity does behave as we would want a productivity measure to behave for HPC programs, i.e., good productivity means lower total effort, lower HPC execution time and higher speed up.

Table 3: Productivity experiment: Game of Life

shows the distribution of programming assignments across different programming models for the first 7 classes (using the same CxAy coding used in Table 2). Multiple instances of the same programming assignment lend the results to meta-analysis to be able to consider larger populations of students.

summarizes the number of times each technology has been applied to each programming problem. More details are given in [14] and [21].

Problem	serial	MPI	OpenMP	Matlab*P	XMT-C	Co-Array Fortran	UPC	Hybrid MPI-OpenMP
Game of life	4	5	2	1		2	2	
SWIM			1					
Buffon-Laplace	2	3	2	3				
Laplace's equation	1	1	1	1				
Sharks & fishes	1	2	2			1		
Grid of resistors	1	1	1	1				
Matrix power via prefix		3	1			1	1	
Sparse conjugate-gradient		2				1	1	
Dense matrix-vector multiply	1	1	1					
Sparse matrix-vector multiply	1	1			2			
Sorting	2	3	1		2			
Quantum dynamics		2						
Molecular dynamics								1
Randomized selection					1			
Breadth-first search					1			
LU decomposition			1					
Shortest path			1					
Search for intelligent puzzles		1						

	Serial	MPI	OpenMP	Co-Array Fortran	StarP	XMT
Nearest-Neighbor Type Problems						
Game of Life	C3A3	C3A3 C0A1 C1A1	C3A3			
Grid of Resistors	C2A2	C2A2	C2A2		C2A2	
Sharks & Fishes		C6A2	C6A2	C6A2		
Laplace's Eq.		C2A3			C2A3	
SWIM			C0A2			
Broadcast Type Problems						
LU Decomposition			C4A1			
Parallel Mat-vec					C3A4	
Quantum Dynamics		C7A1				
Embarrassingly Parallel Type Problems						
Buffon-Laplace Needle		C2A1 C3A1	C2A1 C3A1		C2A1 C3A1	
Other						
Parallel Sorting		C3A2	C3A2		C3A2	
Array Compaction						C5A1
Randomized Selection						C5A2

Table 4: Some of the early classroom experiments on specific architectures

Table 4: Multiple classr

oom studies for each technology

	Serial	MPI	OpenMP	Co-Array Fortran	StarP	XMT
Nearest-Neighbor Type Problems						
Game of Life	C3A3	C3A3 C0A1 C1A1	C3A3			
Grid of Resistors	C2A2	C2A2	C2A2		C2A2	
Sharks & Fishes		C6A2	C6A2	C6A2		
Laplace's Eq.		C2A3			C2A3	
SWIM			C0A2			
Broadcast Type Problems						
LU Decomposition			C4A1			
Parallel Mat-vec					C3A4	
Quantum Dynamics		C7A1				
Embarrassingly Parallel Type Problems						
Buffon-Laplace Needle		C2A1 C3A1	C2A1 C3A1		C2A1 C3A1	
Other						
Parallel Sorting		C3A2	C3A2		C3A2	
Array Compaction						C5A1
Randomized Selection						C5A2

For example, we can use this data to partially answer an earlier stated hypothesis (Hyp. 4: *An MPI implementation will require more code than an OpenMP implementation*). Table 5 shows the relevant data giving credibility to this hypothesis (but this data is not statistically significant yet)

Table 5: MPI program size compared to OpenMP program size

Dataset	Programming Model	Application	Lines of Code
C3A3	Serial	Game of Life	mean 175, sd 88, n=10
	MPI		mean 433, sd 486, n=13
	OpenMP		mean 292, sd 383, n=14
C2A2	Serial	Resistors	42 (given)
	MPI		mean 174, sd 75, n=9
	OpenMP		mean 49, sd 3.2, n=10

1. An alternative parallel programming model is the Parallel Random Access Machine (PRAM) model [22], which supports fine-grained parallelism and has a substantial history of algorithmic theory. Explicit Multi-threading C (XMT-C) [23] is an extension of the C language that supports parallel directives to provide a PRAM-like model to the programmer. A prototype compiler exists that generates code which runs on a simulator for an XMT architecture. We conducted a feasibility study in a class to compare the effort required to solve a particular problem. After comparing XMT-C development to MPI, on average, students required less effort to solve the problem using XMT-C compared to MPI. The reduction in mean effort was approximately 50%, which was statistically significant at the level of $p<.05$ using a t-test [24].
2. While OpenMP generally required less effort to complete (Figure 4), the comparison of defects between MPI and OpenMP, however, did not yield statistically significant results, which contradicted a common belief that shared memory programs are harder to debug. However, our defect data collection was based upon programmer-supplied effort forms, which we know are not very accurate. This led to the defect analysis mentioned previously [25], where we intend to do a more thorough analysis of defects made.

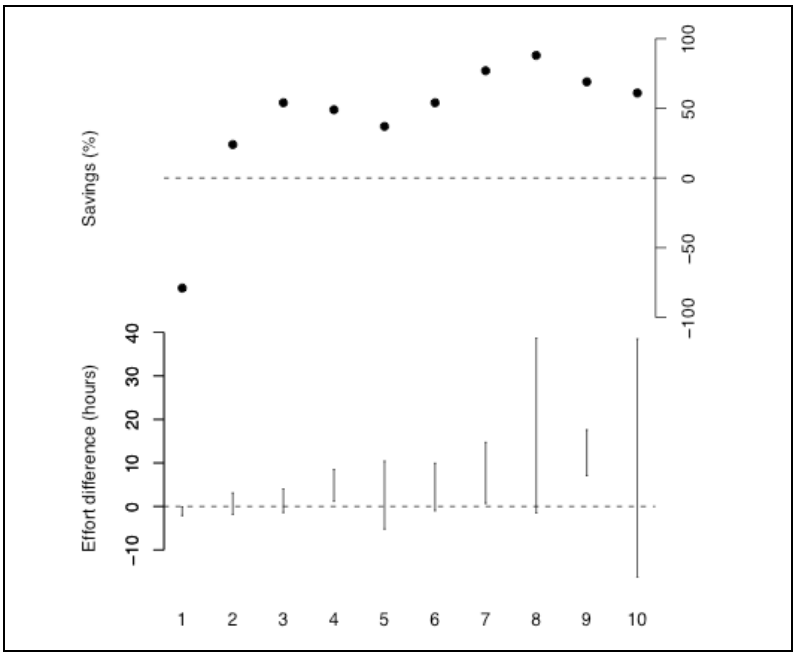

Figure 4: Time saved using OpenMP over MPI for 10 programs. (MPI used less time in only case 1 above)

3. We collected low-level behavioral data from developers in order to understand the "workflows" that exist during HPC software development. A useful representation of HPC workflow could both help characterize the bottlenecks that occur during development and support a comparative analysis of the impact of different tools and technologies upon workflow. One hypothesis we are studying is that the workflow can be divided into one of five states: serial coding, parallel coding, testing, debugging, and optimization.

   In a pilot study at the University of Hawaii in Spring of 2006, students worked on the Gauss-Seidel iteration problem using C and POSIX threads (or PThreads) [17] in a development environment that included automated collection of editing, testing, and command line data using Hackystat. We were able to automatically infer the "serial coding" workflow state as the editing of a file not containing any parallel constructs (such as MPI, OpenMP, or PThread calls), and the "parallel coding" workflow state as the editing of a file containing these constructs. We were also able to automatically infer the "testing" state as the occurrence of unit test invocation using the CUTest tool. In our pilot study, we were not able to automatically infer the debugging or optimization workflow states, as students were not provided with tools to support either of these activities that we could instrument.

   Our analysis of these results leads us to conclude that workflow inference may be possible in an HPC context. We hypothesize that it may actually be easier to infer these kinds of workflow states in a professional setting, since more sophisticated tool support is often available which can help support inferencing regarding the intent of a development activity. Our analyses also cause us to question whether the five states that we initially selected are appropriate for all HPC development contexts. It may be that there is no "one size fits all" set of workflow states, and that we will need to define a custom set of states for different HPC organizations in order to achieve our goals.

   Additional early classroom results are given in [8].

4 TOOL DEVELOPMENT

In order to conduct this research, a series of tools were developed (Figure 5):

1. Websites

1. HPCBugBase.org – Defect database
2. http://hpcs.cs.umd.edu – HPCS Development time website

2 Data collection

3. UMDINST – Shell-level time stamps from compilation and execution
4. Experiment Manager - Collect self-reported effort data
5. Shell Logger – Capture all shell commands
6. Hackystat – Low level time stamps for many tools

3 Data conversion

7. Raw data importer – Import UMDINST data to database
8. DB Sanitizer – Remove privacy data from DB

4 Visualization and analysis

9. Automatic Performance Measurement System – Automatically run scripts of programs.
10. UCSB execution harness – Execute programs under controlled conditions
11. CodeVizard – View source code evolution
12. Data Analyzer – Visualization of UMDINST and Experiment Manager data
13. Activity graph – View workflow information

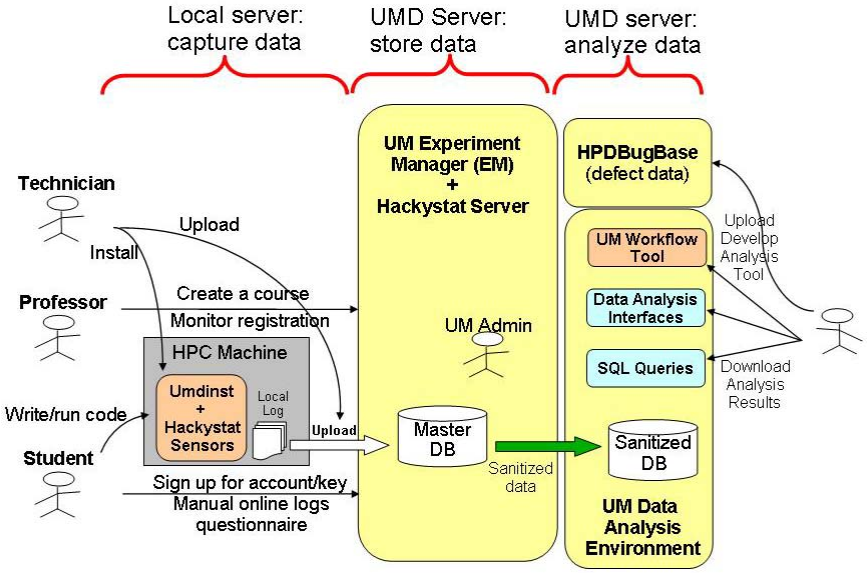

Figure 5: Tool structure

The websites (1.1, 1.2) allow for others to track our research and collect defects via a wiki. The HPCBugBase is described in more details in Section 4.2 below.

The data collection tools (2.3 to 2.6) are focused around the UMD experiment manager, explained in section 4.1 that follows. Hackystat is a product of Philip Johnson of the University of Hawaii, but we have worked with Philip in addressing issues his tool originally didn't process. The data conversion tools (3.7, 3.8) are internal conversion tools within the experiment manager used to handle data from a variety of sources and to handle privacy issues required by federal and state laws. The visualization tools (4.9 through 4.13) are still under development, and the lack of continued funding prevents their completion. A prototype of the data analyzer (tool 4.12) is part of the Experiment Manager explained below.

4.1 Experiment Manager

As stated earlier, we collected effort data from student developments and also collected data from professional HPC programmers in 3 ways: manually from the participants, automatically from timestamps at each system command, and automatically via the Hackystat tool, sampling the active task at regular intervals. All 3 methods provided different values for "effort," and we developed models to integrate and filter each method to provide an accurate picture of effort.

Our collection methods evolved one at a time. To simplify the process of students (and other HPC professionals) providing needed information, we developed an experiment management package (Experiment Manager) to more easily collect and analyze this data during the development process. It includes effort, defect and workflow data, as well as copies of every source program during development. Tracking effort and defects should provide a good data set for building models of productivity and reliability of HPC codes.

We evolved the Experiment Manager framework (Figure 5) to mitigate the complexities described in the previous section. The framework is an integrated set of tools to support software engineering experiments in HPC classroom environments. While aspects of the framework have been studied by others, the integration of all features allows for a uniform environment that has been used in over 25 classroom studies over the past 4 years. The framework supports the following.

1. Minimal disruption of the typical programming process. Study participants solve programming tasks under investigation using their typical work habits, spreading out programming tasks over several days. The only additional activity required is filling out some online forms. Since we do not require them to complete the task in an alien environment or work for a fixed, uninterrupted length of time, we minimize any negative impact on pedagogy or subject overhead.

2. Consistent instruments and artifacts. Use of the framework ensures that the same types of data will be collected and the same types of problems will be solved, which increases confidence in meta-analysis across studies at different universities.

3. Centralized data repository with web interface. The framework provides a simple, consistent interface to the experimental data for experimentalists, subjects, and collaborating professors. This reduces overhead for all stakeholders and ensures that data is consistently collected across studies.

4. Sanitization of sensitive data. The framework provides external researcher with access to the data sets that have been stripped of any information that could identify subjects, to preserve anonymity and comply with the protocols of human subject research as set out by Institutional Review Boards (IRBs) at American universities.

The Experiment Manager has 3 components:

1. *UMD server*: This web server is the entry portal to the Experiment Manager for students, faculty and analysts and contains the repository of collected data.
2. *Local server*: A local server is established on the user machine (e.g., the one used by students at a university) that is used to capture experimental data before transmission to the University of Maryland.
3. *UMD analysis server*: A server stores sanitized data available to the HPCS community for access to our collected data. This server avoids many of the legal hurdles implicit with using human subject data (e.g., keeping student identities private).

   For the near future, our efforts will focus on the following tasks:

   - Evolve the interface to the Experiment Manager web-based tool to simplify use by the various stakeholders (i.e., roles).
   - Continue to develop our tool base, such as the defect data base and workflow models.
   - Build our analysis data base including details of the various hypotheses we have studied in the past.
   - Evolve our experience bases to generate performance measures for each program submitted in order to have a consistent performance and speedup measure for use in our workflow and time to solution studies.

Further details of the Experiment Manager are covered in [18].

4.2 HPCBugBase

A database and corresponding web-based tool has been created to collect and manage defects found in HPC programs. The tool is at http://www.HPCBugBase.org. It is a public wiki allowing for the creation of a defect-based experience base. Figure 6 is the home page of this bug base. Table 6 presents the initial defect classification taxonomy being developed. More information about the HPC Bug Base is found is [25] and [27].

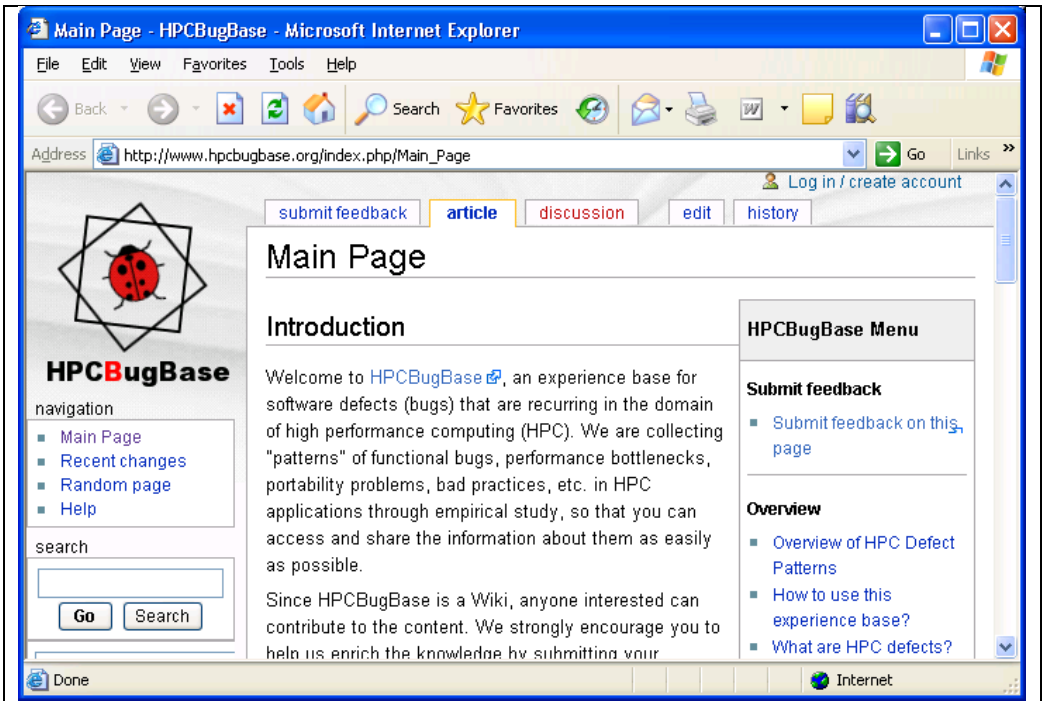

Figure 6: HPCBugBase home page

Table 6: HPC defect classification

Top-level defect type	Sub-types	Brief definition
Use of parallel language features	--	Erroneous use of parallel language features
Space decomposition	--	Incorrect mapping between the problem space and the problem memory space
Side-effect of parallelization (hidden serialization)	I/O hotspots	Serial constructs causing correctness and performance defects when accessed in parallel contexts
	Hidden serialization in library functions	
Synchronization	Deadlock	Incorrect/unnecessary synchronization
	Race	
Performance	Message scheduling	Performance defects in parallel contexts
	Load balancing	
	Failure to exploit locality	
	Excessive synchronization	
	Communication/computation ratio	
	Scalability problems	
	Memory hierarchy	
Memory management	Memory allocation	Inadequate memory management
	Memory cleanup	
Algorithm	--	Program logic not matching the intended purpose of the code
Environment	--	Failure to understand/use code written by other people (library code)

4.3 Summary

During the first three of the four years of this project, we developed a methodology for running HPC experiments in a classroom setting to obtain results we believe are applicable to HPC programming in general. We started to look at larger developments and look at large university and government HPC projects in order to increase the confidence on the early results we obtained with students.

Our development of the Experiment Manager system and HPC bug base allowed us to more easily expand our capabilities in this area. The Experiment Manager permits others to run such experiments on their own in a way that allows for the appropriate controls of the experiment so that results across classes and organization at geographically diverse locations can be compared in order to get a thorough understanding of the HPC development model.

5 THE ASC-ALLIANCE CASE STUDY

Computational scientists face many challenges when developing software that runs on large-scale parallel machines. However, their software development processes have not previously been studied in much detail by software engineering researchers. To better understand the nature of software development in this context, we examined five large-scale computational science software projects, known as the Accelerated Scientific Computing (ASC)-Alliance centers (Section 5.1). We conducted interviews with project leads from all five of the centers to gain insight into the nature of the development processes of large-scale parallel code projects, and to identify issues in the current state-of-the-practice that reduce programmer productivity.

"Computational science" refers to the application of computers to advance scientific research through the simulation of physical phenomena where experimentation would be either prohibitively expensive or simply not possible. Advancement of scientific research depends upon the ability of these scientists to develop software productively, and is therefore of interest to software engineering researchers. However, there are many differences in software development process in this domain compared to other domains such as IT. One major difference is that scientific software can be very computationally demanding and may require the use of the most powerful machines available today, which are sometimes referred to as high-end computing (HEC) systems or supercomputers. These systems present some unique challenges to software development.

To learn more about the process of developing software to run on HPC systems, we studied five different software projects that develop such codes[2]. These projects make up a group of research centers known as the ASC-Alliance centers. Each center owns a large software project which is focused on addressing a different problem in computational science. These centers are provided with access to large-scale HPC systems located at various supercomputing centers.

To study these projects, we conducted interviews with high-level members of each project. These project members were all involved in project management, software architecture, or software integration. Our goals were to characterize product, project organization, and process, both in terms of using the software and developing the software, and to identify particular challenges faced by the developers. Note that we use the terms "developer", "scientist", and "programmer" interchangeably.

5.1 Background

The five ASC-Alliance centers were formed around 1997 by the National Nuclear Security Agency (NNSA, an agency within the Department of Energy) to develop computational simulation as a credible method of scientific research. The major project at each center is focused on solving one particular scientific problem by developing multi-physics, coupled applications. This refers to the simulation of different aspects of physical phenomena (e.g. solid mechanics, fluid mechanics, combustion), which are then "coupled" together to form a single simulation. The five centers are:

- Center for Simulating the Dynamic Response of Materials, California Institute of Technology
    - Goal: simulate the response of materials to strong shocks

[2] In the HEC community, scientific computer programs are referred to as "codes"

- Center for Integrated Turbulence Simulations, Stanford University
    - Goal: simulate full-scale jet engines
- Center for Astrophysical Thermonuclear Flashes, University of Chicago
    - Goal: simulate thermonuclear burn of compact stars
- Center for the Simulation of Advanced Rockets
    - Goal: simulate solid propellant rockets
- Center for Simulation of Accidental Fires & Explosions, University of Utah
    - Goal: simulate fires in contained vessels.

5.2 Goals and methodology

Our goals for conducting this study were to characterize which scientific programming activities are time-consuming and problematic, common problems that scientific programmers face, and the impact of software technologies on developer effort.

We conducted this study within the context of the DARPA High Productivity Computing Systems (HPCS) program. In our earlier work, we ran controlled experiments to evaluate the effect of parallel programming language on programmer effort and program performance, using students from graduate-level parallel computing courses [8]. However, without empirical data on how scientific codes are actually developed, we had no larger context for interpreting our results. In particular, we did not know whether a new parallel programming language would address the major problems that the developers faced, or whether they would adopt a new language if given the opportunity.

To conduct this study, we began by distributing a pre-interview questionnaire to each center that asked for some basic information about the project. Next, we conducted a telephone interview with one or two of the technical leads on the project. From this interview, we generated a summary document, which was sent back to the technical leads for review and corrections.

5.3 Software characteristics

While our main object of study was the software development process of the scientists, we first wanted to characterize the product that they were working on, so we had some context for the software environment that the developers were working in. We asked about *attributes* of the software (information about the size of the codes, degree of code reuse via libraries, organizational structure of the code), and the intended *machine target* (what kinds of machines the codes are intended to run on).

5.4 Attributes

The size of the codes range from 100-500 Kilosource lines of code (KLOC). Most of the codes are written as a mixture of C/C++ and Fortran, with one code being a pure Fortran implementation. One code uses a Python scripting layer that provides an interface for running the application. With one exception, core elements of these projects evolved from pre-existing codes.

All codes make use of the MPI library to achieve parallelism. In addition, each code makes use of external libraries, for features such as I/O, mesh operations – including adaptive mesh refinement, computational geometry, linear algebra, and tools for solving sparse linear systems and systems modeled by partial differential equations.

While these codes use parallel libraries which sit atop MPI, developers were still required write raw MPI code to achieve desired functionality. Therefore, they must deal with the additional complexities that are well-known in writing message-passing applications. Some of the codes use a layered approach which hides the details of message-passing, so that a programmer can add additional functionality without writing MPI code. However, these abstraction layers had to be written from scratch.

Each code is organized into independent subsystems, and the subsystems are maintained by separate individuals or small groups. All codes use a component-based architecture to minimize coupling between individual subsystems. In several of the projects, these independent subsystems are almost like separate projects: they can run as standalone applications and may incorporate new features that are independent of the larger, coupled application. Since almost all of the codes involve multiple programming languages, they must deal with language interoperability issues. (The one exception, a pure Fortran application, once used a Python framework to drive the application but abandoned it because of the difficulty in porting a hybrid Python/Fortran application to multiple platforms). One project built their component framework around the Common Component Architecture (CCA), which is a community effort to simplify the task of building such multi-language, coupled codes. In that case, the chief software architect was an early adopter of this technology and is actively involved with the larger CCA effort. The other projects developed their own communication frameworks.

5.5 Machine target

The codes are designed to run on "flat" MPI-based machines (i.e. all communication takes place through message-passing, even if some processes share physical memory). While all of the codes currently run on clusters of symmetric multiprocessors (SMPs), none of them have been explicitly optimized to take advantage of the SMP nodes: the developers assume that the vendor MPI implementations are efficient enough that optimizing for SMP nodes would not yield large performance improvements. Tuning for a specific architecture is considered a poor use of resources: the investment required to gain expertise in a particular architecture is too great given that new architectures appear every six months.

Two projects experimented in the past with improving performance on clusters of SMPs by using OpenMP to leverage parallelism within nodes and MPI to leverage parallelism across nodes. Results were mixed: one project found that a pure MPI implementation was competitive with hybrid MPI-OpenMP approach, and the other did observe increased performance when incorporating OpenMP but have not had a chance to follow up on this work due to other priorities.

5.6 Project organization

Like all software projects that involve more than a single individual, the developers on these projects must coordinate their efforts. We wanted to understand the *organizational structure* (how the project was organized), the *staff* (who the developers were) and their *configuration management* process (how the developers coordinated to make changes to the code). We were looking for similarities and differences with software projects in other domains, and whether the scientists encountered any domain-specific issues from a project management point of view.

Each project is divided up into groups that focus on different aspects of the problem. This division is reflected in the code, where the software is partitioned into independent subsystems and each subsystem is owned by one of the groups. Each subsystem has one or two chief programmers who understand the subsystem in depth and are responsible for it. These chief programmers make the majority of the changes to the code. Each project also has either a chief software architect or a group who is responsible for the integration code.

Development is compartmentalized, and the groups are relatively independent. There are integrated code development meetings once a week where the core developers from each of the groups meet to discuss issues such as coordinating code changes that will affect more than one module.

5.7 Staff

In total, there are about seventy-five people actively involved on a given project. Ten to twenty-five of these people are core developers that routinely contribute code to the project. The developers consist of professionals, professional staff members with M.S. and PhD degrees, postdocs, and graduate students. Their backgrounds are in physics, chemistry, applied math, engineering (mechanical, civil, aerospace, chemical), and computer science. The experience of the programmers ranges from five to twenty-five years of sequential programming, and zero to fifteen years of parallel programming. The projects also have graduate students who work on the code as part of their research, though they are not core developers.

5.8 Configuration management

The projects use version control systems such as Concurrent Version System (CVS) [28] and subversion to coordinate changes to the code, and all have integrated version control into their development process. No projects have adopted a formal process for approving code before it is checked into the repository. Instead, there is general agreement that test cases should pass before commits are made to the repository. Developers are individually responsible for performing any unit testing, standalone testing, and integration testing that may be necessary. On one project, all developers are automatically notified by email whenever code is checked in to the repository so that the developers are aware of recent modifications that might affect them.

Since all codes are in active use for scientific research and active development, the projects must allow the developers to modify the code while ensuring that a stable version is always available. Therefore, all projects maintain both *stable* and *development* versions of the code.

Only one project has a formal bug-tracking system that is in active use. On the other projects, defect tracking is accomplished through informal communication among project members and through the use of wikis. Some projects have attempted in the past to introduce defect tracking systems, but these systems were not adopted by the developers.

5.9 Software usage

Our study focused mainly on issues related to the development of the software. However, we also wanted to get a sense of how the software was used, and *who was using it*. Since problems with requirements are a major issue in other domains of software engineering, and requirements are often driven by user needs, we wanted to understand the role of the *user* in the software development process in this domain. In addition, we wanted to understand what the *execution times* were like. We did not know just how long these types of programs took to run, and we believed going into the study that large execution times were a major obstacle to programmer productivity. We wanted to understand the entire process of how the software was used, from *setting up the input* to *examining the output*.

5.10 Users

The main users of the codes are research scientists who are the active developers. Some of the users are students who are using the software for their own scientific research, and are not active in the code development, but these efforts are not the primary concern of the Centers. Some codes have found a user base outside of the project. These external users may even modify the program to suit their own needs.

5.11 Execution times

Characterizing the execution times of the codes is difficult because execution times vary enormously depending upon the size of the problem. Typical runs are on the order of ten to one hundred hours of execution time.

5.12 Setting up the input

Most projects use configuration files for specifying program parameters, with two exceptions: one project uses an interactive Python-based scripting interface, and another provides a programmatic Fortran interface for specifying the initial conditions of the simulation. Some projects have expressed interest in developing a graphical interface to simplify the task of setting up the input for a run.

For some projects, generating the inputs is a very time-consuming task. Some of the codes simulate systems with intricate geometries (e.g. the space shuttle), which are modeled as unstructured meshes. Generating the mesh for an input can take an experienced user from half an hour to weeks or months. In one case, a user spent a year generating a mesh for input. Determining whether a given mesh is of sufficient quality is an active area of research.

It can also take hours to weeks to retrieve the physics data needed to run the software, depending on what type of data is needed and how good the existing documentation is. In some cases, determining the correct initial conditions for the simulation is also an active area of research.

5.13 Examining the output

Users apply visualization tools for examining the output of the simulations. The projects use a mix of visualization tools developed in-house and third-party tools.

5.14 Development activities

The developers engage in different activities during the course of development. We asked for details about the following categories: *adding new features* to the code base, *testing* the code to verify correctness, *tuning* the code to improve performance, *debugging* the code to remove defects, and *porting* the code to new platforms.

5.15 Adding new features

Each year, the centers plan on running a major set of simulations. These simulations drive an implementation plan which determine what new features need to be added to the software. Scientists can explore avenues of research, but the overall direction is set by the implementation plan. For centers with larger external user bases, sufficiently strong demand from outside users can also drive the addition of new features.

New features added to the codes can be classified into two categories: those that are localized within an individual subsystem (low-level change), or those that involve changes across sub-systems (high-level change). Low-level changes are administered solely by the owners of the subsystem being modified and require no communication across groups. High-level changes require some degree of coordination.

Since the projects have been in operation for almost a decade, the code bases are all fairly mature and are currently being applied to do real science. Very few new subsystems are planned for development, although there continue to be enhancements to existing subsystems. Most modules have satisfactory parallel performance, with the exception of very new modules and modules where efficient parallelization is still an open research problem (e.g. adaptive mesh refinement).

In some of the projects, the developers do not have to write code explicitly in parallel but instead build on top of a parallel infrastructure that abstracts away the parallelization details. Other projects require that the developers program directly to the MPI library.

5.16 Testing

All projects use a suite of regression tests to catch any errors introduced by programmers modifying the code. Some projects have an automated system for running regression tests, and others run the regression tests manually. One project requires that new students who check out the code are required to run the regression tests as part of their learning process.

Testing a new algorithm is a challenging task in this environment. It is not sufficient to define simple test cases where modules are fed known inputs and checked against expected outputs. Rather, algorithms are evaluated in terms of qualities such as stability, accuracy, speed, and linear scalability. A module is considered to be functioning correctly if, for the class of inputs that are of interest, the quality of the module's output is sufficient to allow it to be coupled with other modules and produce coupled applications. Since the inputs of interests change over time as more complex simulations are attempted, an algorithm that is acceptable today may not be acceptable tomorrow. Therefore, the testing process is different from other software domains because the focus is on identifying *algorithmic defects* (i.e. evaluating the quality of the algorithm) rather than on *coding defects* (i.e. errors in implementing the algorithm in the source code). Finding and fixing *algorithmic defects* is much more challenging than finding and fixing *coding defects*.

Testing the quality of algorithms involves qualitative analysis about how the algorithm behaves. There are different strategies for testing an algorithm, depending upon the nature of the problem (e.g. checking if certain quantities are exactly or approximately conserved, checking if symmetry properties hold, checking against known analytical solutions). Some of the projects work with numerical analysts who can provide mathematical guarantees about certain aspects of the code such as stability, or that certain positive quantities such as energy cannot diverge.

In general, the developers do not know whether an algorithm solves an equation correctly until certain requirements are passed. For example, a module may appear to be performing correctly in isolation, but when used in a coupled application it may behave in unexpected ways. This testing process is very interactive and requires a substantial amount of effort and expertise. Since many of the developers are postdocs and graduate students without extensive experience, the testing process involves a lot of guidance from senior people who understand the broader scope of the physics and software.

5.17 Tuning

Tuning activity occurs when the developers discover that the software is executing much slower than expected. Tuning may be required when the software is being ported to a new platform (see also the *porting* section below), or if there major changes to the software architecture have caused performance penalties, or simply because of changes made to a particular subsystem create a bottleneck. At least one project uses tuning specialists: developers who are skilled at identifying and fixing performance bottlenecks. One particular tuner comes from a local computer science group that develops performance analysis tools.

Since one of the project goals is to develop algorithms that will last across many machine lifetimes, it is not seen as productive to try to maximize the performance on any one particular platform. Instead, code changes are made that will improve performance on a wide range of platforms. In addition, on at least one project the codes are constantly in a state of flux, as new algorithms are continually being evaluated, which involves changing the core components of the code. If there were many machine-specific optimizations in the code, it would be much more difficult to understand the code, which would increase maintenance effort.

For a given application, there is often a considerable amount of tuning that needs to be done in order to achieve reasonable performance on a new platform. This tuning process is mostly about determining data set size, number of processors, and which processors should be assigned which tasks. While the individual projects do not focus on maximizing performance on any one system, they are occasionally able to take advantage of a team of third-party experts who can achieve a large speedup on a particular system.

Developers do use externally developed profiling tools. However, on some of the codes, external profiling tools have failed because they could not deal with an application written in multiple languages. In addition, some of the codes contain their own profiling routines. Some developers find these tools useful, but others say that they are familiar enough with the code that these profiling tools do not reveal any information that is not already known.

There are ongoing efforts to improve performance through the development of new algorithms (e.g. new adaptive mesh refinement algorithms). However, the developers view these efforts as new functionality rather than tuning.

5.18 Debugging

All projects mentioned make some use of TotalView [29], which is a popular parallel debugger. Sequential debugging tools (e.g., Purify [30]) are also used, although they are only useful if the failure can be reproduced when running the program on a single processor. The use of trace statements to debug is common across all projects, although these are difficult to interpret when the program is running on a large number of processors. The developers also examine the simulation outputs with visualization tools to help identify defects.

Developers described several usage patterns for applying the tools to localize defects. One common pattern is to use a debugger to produce a stack trace, which is then used to determine where to insert print statements into the code. Another common debugging pattern is to try to reproduce a defect when running the program on a smaller simulation, as tools such as TotalView and the GNU Project Debugger (gdb) [31] can used very effectively on a moderate number of CPUs.

There are some types of defects for which the existing tools do not provide any additional assistance. Most of the debugging time is spent in algorithmic debugging (i.e. dealing with *algorithmic defects*, see the section on *testing*, above), a process for which typical debugging tools are not appropriate.

Large machines are generally batch-scheduled, which makes debugging more difficult. Since the debugging process typically involves frequent re-running of the code, running under a batch queue can become a week-long process where it would be a matter of hours on a dedicated machine where jobs could be run interactively. While not available during the week the National Labs make available dedicated weekends to the ASC-Alliance centers where a fraction or the entire platform of a particular machine is available for interactive use for a weekend. There are a few of these weekends each year, and about 60 hours of total compute time are provided to the center, which provides sufficient time to run very large jobs and do large-scale debugging. Debugging runs that involve smaller number of processors are done internally.

5.19 Porting

Porting activity most commonly occurs when the DOE purchases a new machine that becomes available for project use. Additionally, porting work may be necessary due to a software upgrade on a system that is already in use. The time required to port to a new system ranges from a single day to several weeks depending on the individual code and the maturity of the development tools on the new platform.

Porting requires a non-trivial amount of effort because, when a new platform is released, the code cannot simply be recompiled and run. Much of the porting effort is spent on a large number of small details that need to be addressed. For example, the build scripts (e.g. makefiles) need to be modified to accommodate the differences in development tools on the new system. Immature compilers on new systems are another source of porting effort; one project spent a year getting the code to run on one particular machine because of C++ issues; while some problems had workarounds, others required that they wait for the vendor to fix the compiler. One project completely abandoned the use of Python to reduce porting effort.

Although porting is only a small percentage of the total effort, it can involve a great deal of work in a very focused time (e.g. several people working for a month). Porting is perceived as a task that involves an unwarranted amount of effort. Some projects have broken compilers and MPI codes on every platform. Two projects did not port their code to Sandia Lab's ASCI Red[3] because the Fortran compiler did not support needed features properly, and it was not worth the effort to try and work around these problems to get the code to run on the machine.

5.20 Effort distribution and bottlenecks

Developers report spending most of their development time (75-95%) on adding new features and testing the system. Tuning and porting take up relatively little of the total development time (1-10%). However, the distribution of effort varies depending on the development phase of the project. Earlier on in the projects, when new data structures had to be designed to handle a particular class of physics problems, effort distribution was different than currently where the effort on data structures is oriented towards modifying them for new areas. The projects occasionally undergo large-scale re-architecting to better incorporate new features, and this also changes the effort distribution.

[3] http://www.sandia.gov/ASCI/Red/

Verification and validation was reported as a common bottleneck. In addition, the debugging of parallel algorithms was a bottleneck across all projects. Such debugging is made more challenging because of the defects that only manifest themselves in more complex (parallel) execution environments where it is more difficult for the programmer to observe what is happening. For example, a code that runs perfectly well on 32 processors, may fail on 64 processors due to interactions caused by the increasing number of parallel interfaces needing to be supported. Or, a subsystem may work well when executing independently, but does not behave as expected when interacting with another subsystem.

Other bottlenecks include:

- Generating input (Computer Aided Design (CAD) modeling, mesh generation)
- Expressing algorithms in parallel
- Performing production runs (obtaining time to run on large machines)
- Understanding old code (when re-architecting)

5.21 General observations

Based on the interview responses, we made some general observations about software engineering in this domain. One surprise to us was the challenging nature of performing *validation* of this type of software. We were also very interested in the role that MPI played in these projects, because of the goals of the HPCS project to develop alternative parallel programming languages. In particular, we were interested in understanding how likely an alternative to MPI, either in the form of a *library or framework* that encapsulates MPI or in the form of another *parallel programming language* could replace the current dominance of MPI in such HPC environments. Finally, we were interested in the scientists' opinions on what *productivity* means to them, since one of our ultimate goals as software engineering researchers is to help improve programmer productivity.

5.22 Validation

Validation of the codes (checking the simulation accurately simulates the phenomena of interest) is a formidable challenge. A validation study is typically a research project or the essence of a thesis. A student will choose a problem that has an interesting set of experimental data associated and then identifies to what extent it can be simulated. Because of the effort involved in running such studies, it is not feasible to validate every new algorithm with an experiment.

5.23 MPI

All projects make extensive use of MPI to achieve parallelism. MPI is a standard library that is efficiently implemented on all types of platforms: It is the only technology that can run 10,000 processor jobs, and the only technology available that will allow the program to be run on a workstation as well as a world-class machine without having to tweak the code in significant ways. In addition, the algorithms employed by the projects all lend themselves to domain decomposition.

Despite MPI being the most appropriate technology for the job, there was widespread dissatisfaction with it: one project member referred to it as “nothing more than high-level assembly language”, where the programmer is responsible for all memory management and process signaling. The additional work required by MPI was identified as a large barrier to productivity for a project starting from scratch. However, one project member mentioned that exposing these low-level details to the programmer is an advantage of MPI, because it gives the programmers more control over what happens in the code.

One of the advantages of MPI is that it is possible to write MPI code knowing very few functions, so it is very easy to teach. While teaching people to use MPI is not particularly hard, teaching people to write MPI effectively is extremely difficult: this fact distinguishes the first year graduate student from the developers who have been at a center for four-to-five years.

5.24 Alternative to MPI: libraries/frameworks

All of the projects used libraries that were built on top of MPI. However, while such libraries can abstract away some of the low-level details of message-passing code, none of the projects adopted such libraries to the extent that they could entirely avoid writing MPI code. Even the libraries that are used can be problematic: on one project, the most troublesome code is the parallel HDF5[4] I/O library that sits atop MPI. Other projects outside of the ASC-Alliance have attempted to reuse existing class libraries to completely abstract away the low-level MPI details, but this has led to problems. For example, on one such project, some C++ code from a parallel framework was not portable across platforms because of differences in compiler implementations. Another obstacle to reuse is that these frameworks make assumptions about how the work will be done, and violating these assumptions requires that the programmer become deeply involved in framework details. The effort to use the framework is roughly the same as the effort involved in writing the code using MPI, which eliminates the benefit of use. Because of this, each project chose to develop a custom framework.

5.25 Alternative to MPI: other languages

There was no alternative to MPI that any of the developers were aware of that could better meet their current needs. Some projects had previously looked at High-Performance Fortran and hybrid MPI-OpenMP as alternative models, but none felt compelled to switch.

Developers mentioned the following desirable features in an alternative parallel programming language

- Hides memory operations from the programmer
- Provides the programmer with a single memory space
- Supports remote puts and gets
- Is easy to use
- Is efficiently implemented on all types of platforms
- Can do everything MPI can do plus provide additional useful capabilities

[4] http://www.hdfgroup.org/HDF5/

Even if a new language met the above criteria, this would not guarantee adoption by the projects. The existing codes are already highly scalable and portable, and have taken years to develop and verify. All of the developers would have to be convinced that the new language would make their programs much better.

Both technical and sociological issues create obstacles for the adoption of a new language. Technical issues include *expressiveness, performance, support for large projects,* and *portability*. A new language would need to be *expressive* enough to support all of the needed algorithmic features (e.g. grid-based data structures, particle-based data structures). A parallel version of C or Fortran would not be considered expressive enough, as the code would probably be no cleaner than it is currently. The *performance* would have to be competitive with C. C++ is an appealing language because C++ code can be made to look like C to achieve better performance. To *support large projects*, the language must support separation of concerns so that developers can work on isolated pieces (some languages assume the programmer has a global view of the system). Finally, it must be *portable*. The developers would have to be convinced that it could be ported to anything else in the future. If the developers had to re-design the parallelism when a new platform comes out, then it won't even be considered. Even C++ is considered a somewhat risky technology because of the varying levels of support for C++ features across vendor compilers.

The main sociological issue is that of *market share*. The developers would have to be confident that the new language would last well into the future. Unfortunately, this creates a chicken-and-egg problem for new languages. Some of the current codes are so complex that it would be very strenuous to migrate to something else. Even if an ideal solution were found, it would not be adopted immediately. The developers would have to experiment with the technology first. They would also like to see a number of large workhorse applications converted and benchmarked. Being able to use the existing code base in some fashion would make the transition easier, although they would still consider a new language that did not allow this, if it satisfied their other requirements.

5.26 Measures of productivity

The developers suggested several measures of productivity that would be useful in their context. One was *scientifically useful results* over calendar time, where *scientifically useful results* implies sufficient simulated time and resolution, plus sufficient accuracy of the physical models and algorithms. Another was *problem size* over calendar time, where the goal is to run a larger simulation in a smaller calendar time. One developer gave the example of running a particular simulation in a calendar week instead of a calendar quarter. This is partially related to the speed of the code, and partially to the availability of the machine.

Other suggested measures had to do with time to implementation: minimize the time between conception of an algorithm and the actual implementation on a parallel architecture, or the time to make modifications to the code (minor or substantial). For users to be productive, the holy grail is to very quickly go from equations to mapping the equations to a parallel model, and then getting an implementation and running it on a machine. Even a petascale machine would be too slow for some of the problems that have been addressed (e.g., cryptography constants revolves around the creation of longer cryptographic keys and faster computers to break those codes). Most of their time is spent trying to map the solution to what is achievable given today's computing resources, so that the problem is solvable in a reasonable period of time.

Therefore, they are focused on the development of more efficient algorithms. They are more concerned with how long it will take to develop a solution than how long a particular run takes. The projects have a much longer-term view on performance issues than HEC vendors.

For the professor whose job it is to turn out students, one suggested metric was the length of time from when a grad student finishes the second year of coursework to when she is a productive researcher in the group. This involves acquiring skills as a developer, as a designer of parallel algorithms, understanding the physics and how parallelism applies to it. At the university, there are considerably more educational aspects to the notion of productivity than in a government or industrial lab. (Additional comments on education are addressed later in Section 6.2.)

Each project had to build up a software infrastructure, largely from the ground up. This required a substantial amount of effort, and it would be more productive not to have to build such infrastructure. One developer related how, after a computational scientist would design and program an algorithm, the program would be handed off to an applications programmer who would then rewrite it so it would run quickly on the system. One developer noted that a good test of a productive environment is when, in such an environment, the applications programmer does not need feel the need to rewrite the computational scientists' programs.

Some productivity issues were only tangentially related to the details of expressing parallelism in the code. One developer brought up the importance of doing verification and validation to the productivity of the project, especially given the substantial learning curve involved in adopting a formal verification and validation process, which was pushed by the National Laboratories several years ago. Other productivity issues mentioned were the ability to estimate the effort required to implement features and the resultant performance of the code, the degree of code reuse by outside users, and the ease of which external users can modify the code to suit their own needs.

5.27 Conclusions to case study

We have attempted to capture different aspects of the software development process for large-scale HEC applications, focusing on areas that are particularly challenging. These projects face some unique challenges, both because of the nature of the problem, and because of the difficulties associated with programming the environment. Note that because all the projects that we examined are based in academic environments, it is unclear how much would generalize to computational science projects in industry or government.

6 UNDERSTANDING THE HIGH-PERFORMANCE COMPUTING COMMUNITY

In addition to the ASC case study of the previous section, for the past few years, we've had the opportunity, as software engineers, to observe the development of computational-science software (called *codes*) built for high-performance-computing (HPC) machines in many different contexts. Although we haven't studied all types of HPC development, we've encountered a wide cross-section of projects. Despite these projects' diversity, several common traits exist:

- **Many developers receive their software training from other scientists. Although the scientists often have been writing software for many years, they generally lack formal software engineering (SE) training, especially in managing multiperson development teams and complex software artifacts.**
- **Many of the codes aren't initially designed to be large. They start small and then grow on the basis of their scientific success.**
- **Many development teams use their own code (or code developed as part of their research group).**

For these reasons (and many others), development practices in this community differ considerably from those in more "traditional" SE.

We aim here to distill our experience about how software engineers can productively engage the HPC community. Several SE practices generally considered good ideas in other development environments are quite mismatched to the HPC community's needs. For SE researchers, the keys to successful interactions include a healthy sense of humility and the avoidance of assumptions that SE expertise applies equally in all contexts.

A list of the 500 fastest supercomputers (www.top500.org) shows that, as of November 2007, the most powerful system had 212,992 processors. Although a given application wouldn't routinely use all these processors, it would regularly use a high percentage of them for a single job. Effectively using tens of thousands of processors on a single project is considered normal in this community.

We were interested in codes requiring nontrivial communication among the individual processors throughout the execution. Although HPC systems have many uses, a common application is to simulate physical phenomena such as earthquakes, global climate change, or nuclear reactions. These codes must be written to explicitly harness HPC systems' parallelism. Although many parallel-programming models exist, the dominant model is MPI (message-passing interface), a library where the programmer explicitly specifies all communication. Fortran remains widely used for developing new HPC software, as do C and C++. Frequently, a single system incorporates multiple programming languages. We even saw several projects use dynamic languages such as Python to couple different modules written in a mix of Fortran, C, and C++.

The HPCS program's significance was its shift in emphasis from execution time to time-to-solution, which incorporates both development and execution time. We began this research by running controlled experiments to measure the impact of different parallel-programming models. Because the proposed languages weren't yet usable, we studied available technologies such as MPI, OpenMP, Unified Parallel C (UPC), Co-Array Fortran, and Matlab*P, using students in parallel-programming courses from eight different universities.

To widen this research's scope, we collected "folklore"—that is, the community's tacit, unformalized view of what's true. We collected it first through a focus group of HPC researchers, then by surveying HPC practitioners involved in the HPCS program, and then by interviewing a sampling of practitioners including academic researchers, technologists developing new HPC systems, and project managers. Finally, we conducted case studies of projects at both US government labs and academic labs.

To understand why certain software engineering technologies are a poor fit for computational scientists, it's important to first understand the scientists' world and the constraints it places on them. Overall, we found that there's no such thing as a single "HPC community." Our research was restricted entirely to computational scientists using HPC systems to run simulations. Despite this narrow focus, we saw enormous variation, especially in the kinds of problems that people are using HPC systems to solve. Table 7 shows four of the many attributes that vary across the HPC community.

Table 7: HPC community attributes

Attribute	Values	Description
Team size	Individual	This scenario, sometimes called the "lone researcher" scenario, involves only one developer.
	Large	This scenario involves "community codes" with multiple groups, possibly geographically distributed.
Code life	Short	A code that's executed few times (for example, one from the intelligence community) might trade less development time (less time spent on performance and portability) for more execution time.
	Long	A code that's executed many times (for example, a physics simulation) will likely spend more time in development (to increase portability and performance) and amortize that time over many executions.
Users	Internal	Only developers use the code.
	External	The code is used by other groups in the organization (for example, at US government labs) or sold commercially (for example, Gaussian, www.gaussian.com)
	Both	"Community codes" are used both internally and externally. Version control is more complex in this case because both a development and a release version must be maintained.

One possible measure of productivity is scientifically useful results over calendar time. This implies sufficient simulated time and resolution, plus sufficient accuracy of the physical models and algorithms. (All quotes are from interviews with scientists unless otherwise noted.)

[Floating-point operations per second] rates are not a useful measure of science achieved.—user talk, IBM scientific users group conference

Initially, we believed that performance was of paramount importance to scientists developing on HPC systems. However, after in-depth interviews, we found that scientific researchers focus on producing publishable results. Writing codes that perform efficiently on HPC systems is a means to an end, not an end in itself. Although this point might sound obvious, we feel that many in the HPC community overlook it.

A scientist's goal is to produce new scientific knowledge. So, if scientists can execute their computational simulations using the time and resources allocated to them on the HPC system, they see no need for or benefit from optimizing the performance. They see the need for optimization only when they can't complete the simulation at the desired fidelity with the allocated resources. When optimization is necessary, it's often broad-based, not only including traditional computer science notions of code tuning and algorithm modification but also rethinking the underlying mathematical approximations and potentially fundamentally changing the computation. So, technologies that focus only on code tuning are of somewhat limited utility to this community.

Computational scientists don't view performance gains in the same way as computer scientists. For example, one of us (trained in computer science) improved a code's performance by more than a factor of two. He expected this improvement would save computing time. Instead, when he informed the computational scientist, the scientist responded that the saved time could be used to add more function—that is, to get a higher-fidelity approximation of the problem being solved.

Conclusion: Scientists make decisions based on maximizing scientific output, not program performance.

Performance vs. portability and maintainability

If somebody said, maybe you could get 20 percent [performance improvement] out of it, but you have to do quite a bit of a rewrite, and you have to do it in such a way that it becomes really ugly and unreadable, then maintainability becomes a real problem. … I don't think we would ever do anything for 20 percent. The number would have to be between 2x and an order of magnitude. … Readability is critical in these codes: describe the algorithms in a mathematical language as opposed to a computer language.

Scientists must balance performance and development effort. We saw a preference for technologies that let scientists control the performance to the level needed for their science, even by sacrificing abstraction and ease of programming. Hence their extensive use of C and Fortran, which offer more predictable performance and less abstraction than higher-level programming languages.

Conversely, the scientists aren't driven entirely by performance. They won't sacrifice significant maintainability for modest performance improvements. Because the codes must run on multiple current and future HPC systems, portability is a major concern. Codes must run efficiently on multiple machines. Application scientists aren't interested in performing machine-specific performance tuning because they'll lose the benefits of their efforts when they port the code to the next platform. In addition, source code changes that improve performance typically make code more difficult to understand, creating a disincentive to make certain kinds of performance improvements.

Conclusion: Scientists want the control to increase performance as necessary but won't sacrifice everything to performance.

Verification and validation for scientific codes

Testing is different. … It's very much a qualitative judgment about how an algorithm is actually performing in a mathematical sense. … Finally, when the thing is working in a satisfactory way—say, in a single component—you may then go and run it in a coupled application, and you'll find out there are some features you didn't understand that came about in a coupled application and you need to go back and think about those.

Simulation software commonly produces an approximation to a set of equations that can't be solved exactly. You can think of this development as a two-step process: translating the problem to an algorithm and translating the algorithm to code. You can evaluate these approximations (mapping a problem to an algorithm) qualitatively on the basis of possessing desirable properties (for example, stability) and ensuring that various conservation laws hold (for example, that energy is conserved). The approximation's required precision depends on the nature of the phenomenon you're simulating. For example, new problems can arise when you integrate approximations of a system's different aspects. Suddenly, an approximation that was perfectly adequate for standalone use might not be good enough for the integrated simulation. Identifying and evaluating an algorithm's quality is a challenge. One scientist we spoke with said that algorithmic defects are much more significant than coding defects.

Validating simulation codes is an enormous challenge. In principle, you can validate a code by comparing the simulation output with a physical experiment's results. In practice, because simulations are written for domains in which experiments are prohibitively expensive or impossible, validation is very difficult. Entire scientific programs, costing hundreds of millions of dollars per year for many years, have been built around experimental validation of large codes.

Conclusion: Debugging and validation are qualitatively different for HPC than for traditional software development.

Skepticism of new technologies

I hate MPI, I hate C++. [But} if I had to choose again, I would probably choose the same.

Our codes are much larger and more complex than the "toy" programs normally used in [classroom settings]. We would like to see a number of large workhorse applications converted and benchmarked.

The scientists have a cynical view of new technologies because the history of HPC is littered with new technologies that promised increased scientific productivity but are no longer available. Some of this skepticism is also due to the long life of HPC codes; frequently, a code will have a 30-year life cycle. Because of this long life, scientists will embrace a new technology only if they believe it will survive in the long term. This explains MPI's widespread popularity, despite constant grumbling about its difficulty.

Scientific programmers often develop code such that they can plug in different technologies to evaluate them. For example, when MPI was new in the 1990s, many groups were cautious about its long-term prospects and added it to their code alongside existing message-passing libraries. As MPI became widely used and trusted, these older libraries were retired. Similar patterns have been observed with solver libraries, I/O libraries, and tracing tools.

The languages being developed in the DARPA HPCS program were intended to extend the frontiers of what's possible in today's machines. So, we sought practitioners working on very large codes running on very large machines. Because of the time they've already invested in their codes and their need for long-lived codes, they all expressed great trepidation at the prospect of rewriting a code in [porting normally refers to a new platform, rewriting to changing languages] a new language.

Conclusion: A new technology that can coexist with older ones has a greater chance of success than one requiring complete buy-in at the beginning.

Shared, centralized computing resources

The problem with debugging, of course, is that you want to rerun and rerun. The whole concept of a batch queue would make that a week-long process. Whereas, on a dedicated weekend, in a matter of hours you can pound out 10 or 20 different runs of enormous size and understand where the logic is going wrong.

Because of HPC systems' cost, complexity, and size, they're typically located at HPC centers and shared among user groups, with batch scheduling to coordinate executions. Users submit their jobs to a queue with a request for a certain number of processors and maximum execution time. This information is used to determine when to schedule the job. If the time estimate is too low, the job will be preemptively terminated; if it's too high, the job will wait in the queue longer than necessary.

Because these systems are shared resources, scientists are physically remote from the computers they use. So, potentially useful tools that were designed to be interactive become excessively slow and are soon discarded because they don't take into account the long latency times of remote connections. Unfortunately for scientists, using an HPC system typically means interacting with the batch queue.

Debugging batch-scheduled jobs is also tedious because the queue wait increases the turnaround time. Some systems provide "interactive" nodes that let users run smaller jobs without entering the batch queue. Unfortunately, some defects manifest themselves only when the code runs on large numbers of processors.

Center policies that use system utilization as a productivity metric might exacerbate the problem of the queue. Because utilization is inversely proportional to availability, policies that favor maximizing utilization will have longer waits. As a counterexample, Lincoln Laboratories provides interactive access to all users and purchases excess computing capacity to ensure that users' computational needs are met.

Conclusion: Remote access precludes the use of certain software tools, and system access policies can significantly affect productivity.

6.1 Mismatches between computational science and SE

Repeatedly, we saw that SE technologies that don't take the scientists' constraints into account fail or aren't adopted. The computer science community isn't necessarily aware of this lesson. Software engineers collaborating with scientists should understand that the resistance to adoption of unfamiliar technologies is based on real experiences. For example, concepts such as the Capability maturity Model Integration (CMMI) [32] aren't well matched to the incremental nature of HPC development.

Object-oriented languages

Java is for drinking.—parallel-programming course syllabus

Developers on a project said, "We're going to use class library X that will hide all our array operations and do all the right things." … Immediately, you ran into all sorts of issues. First of all, C++, for example, was not transportable because compilers work in different ways across these machines.

Object-oriented (OO) technologies are firmly entrenched in the SE community. But in the HPC community, C and Fortran still dominate, although C++ is used and one project was exploring the use of Java. We also saw some Python use, although never for performance-critical code.

Fortran-like Matlab has seen widespread adoption among scientists, although not necessarily in the HPC community. To date, OO hasn't been a good fit for HPC, even though the community has adopted some concepts. One reason for the lack of widespread adoption might be that OO-based languages such as C++ have been evolving much more rapidly than C and Fortran in recent years and are therefore riskier choices.

Conclusion: More study is needed to identify why OO has seen such little adoption and whether pockets exist in HPC where OO might be suitable.

Frameworks

If you talk about components in the Common Component Architecture or anywhere else, components make very myopic decisions. In order to achieve capability, you need to make global decisions. If you allow the components to make local decisions, performance isn't as good.

Frameworks provide programmers a higher level of abstraction, but at the cost of adopting the framework's perspective on how to structure the code. Example HPC frameworks include

- Parallel Object-Oriented Methods and Applications (Pooma), a novel C++ OO framework for writing parallel codes that hides the low-level details of parallelism, and
- Common Component Architecture (CCA), for implementing component-based HPC software.

Douglass Post and Richard Kendall tell how Los Alamos National Laboratory sought to modernize an old Fortran-based HPC code using Pooma. Even though the project spent over 50 percent of its code-development resources on Pooma, the framework was slower than the original Fortran code. It also lacked the flexibility of the lower-level parallel libraries to implement the desired physics.

The scientists in our studies don't use frameworks. Instead, they implement their own abstraction levels on top of MPI to hide low-level details, and they develop their own component architecture to couple their subsystems.

Of all the multi-physics applications we encountered, only one used any aspect of CCA technology, and one of that application's developers was an active member of the CCA initiative. When we asked scientists about the lack of reuse of frameworks such as Pooma, they responded that such frameworks force them to adapt their problem to the interface supported by the framework. They feel that fitting their problem into one of these frameworks will take more effort than building their own framework atop lower-level abstractions such as MPI.

For many frameworks, a significant barrier to their use is that you can't integrate them incrementally. As we noted earlier, a common risk-mitigation strategy is to let competing technologies coexist with a code while they're under evaluation. However, the nature of many frameworks makes this impossible.

Conclusion: Scientists have yet to be convinced that reusing existing frameworks will save them more effort than building their own from scratch.

Integrated development environments

Integrative Development Environments (IDEs) try to impose a particular style of development on me, and I am forced into a particular mode.—US government laboratory scientist

We saw no use of integrated development environments (IDEs) such as Eclipse because they don't fit well into the typical workflow of a scientist running a code on an HPC system. For example, IDEs have no facilities for submitting jobs to remote HPC queues. IDEs also don't support debugging and profiling for parallel machines. The Eclipse Parallel Tools Platform project (www.eclipse.org/ptp) is attempting to provide this functionality.

In addition, although Eclipse supports HPC languages such as Fortran and C/C++, they're second-class citizens in the Eclipse ecosystem, which focuses on Java-related technologies. Whether the larger HPC-system community will adopt these technologies is an open question.

Conclusion: Unless IDEs support remote execution on batch-queued systems, HPC practitioners won't likely adopt them.

But well-matched technologies are adopted

We're astrophysicists, which seems to mean we disdain good software engineering practices until we get bit … hard … >10 times. Nevertheless, we are starting to learn the importance of source control, regression testing, code verification, and more.

We were using CVS until a few months ago. Now we migrated to Subversion. We've had version control since day 1.

FlashTest, the tool for nightly regression testing of FLASH, has been generalized to be usable with any code that uses steps similar to FLASH in building.

Roccom is an innovative object-oriented, data-centric integration framework developed at CSAR [the Center for Simulation of Advanced Rockets] for large-scale numerical scientific simulation.

Scientists do embrace some SE techniques and concepts, when they're a good fit. Every multi-developer project we encountered used a version control system such as CVS or Subversion to coordinate changes. We also saw some use of regression-testing methods, including tests across platforms and compilers. We saw extensive reuse-in-the-small, in the form of reusing externally developed libraries such as preconditioners, solvers, adaptive mesh refinement support, and parallel I/O libraries.

On multi-physics applications involving integration of multiple models maintained by independent groups, the scientists devoted much effort on software architecture for integrating these components, including using OO concepts. In one case, they explicitly used an OO language, C++. In another case, they implemented an OO architectural framework using a non-OO language—Fortran 90.

Conclusion: Scientists working on large projects see the value of an architectural infrastructure, but they're more disposed to build their own.

6.2 What SE can do to help scientists

So, how can we software engineers best apply our knowledge and talents to assist the computational-science community?

Practices and processes

In our study of existing literature, our software environment is not entirely unique. However, our desire to provide an environment that supports development from the inception of high-risk, high-payoff mathematical software to eventual production quality tools is unusual.

We're doing a loose version of Extreme Programming or agile.

As software practitioners, we're familiar with how the software development process affects productivity and quality. We can help in two ways.

First, we can tailor and transfer existing SE practices to the HPC community. We know from observation that larger projects have successfully adopted some mainstream practices. It's important to publicize these successes. Other SE practices, such as inspections, could be successfully adapted but haven't been. Inspections are important here because of the challenges of verification and validation, but they must be tailored for this domain. As always, it's important to take into account this community's environment and constraints to avoid mismatches such as the ones we've mentioned.

Second, we can help capture and disseminate computational-science-specific practices that have been successfully adopted.

Education

Education and outreach to create code that is parallelized—#1 user priority.

Teaching people to use MPI is not very hard. Writing MPI programs that perform well is extremely difficult. That's the difference between a first-year grad student and someone who has been at the center for four to five years.

For the professor whose job is to turn out students, the correct metric is, how long does it take to take a grad student who just finished, say, their second year of coursework to be a productive researcher in the group? That involves a lot more than just actual runtime on the machine. It involves time picking up the skills to be a successful developer, picking up skills as a designer of parallel algorithms, picking up enough physics to understand the problem he's solving and how parallelism applies to it.

We observed both parallel-programming classes and HPC practitioners in action. Although students learned the basic principles of HPC development in their courses, they weren't properly prepared for the kind of software development they needed to do. So, there was a long learning curve to becoming productive, involving the apprenticeship model of working closely with more experienced practitioners.

At the university level, we can develop SE courses specifically for computational scientists. We can also work toward other models of disseminating SE knowledge in this domain. For example, to improve the quality of students' assignments, we've developed materials for teaching them about HPC defects (www.hpcbugbase.org).

Reuse in the large

A lot of our project is getting all this infrastructure put together that we didn't have [before] and doing this from the ground up. A productive thing would be not to have to do that.

Although we could have painted a gloomy picture here about the prospects of large-scale reuse of frameworks, we believe these technologies could reduce programmer effort significantly. A framework that's built on well-supported technologies such as MPI will have fewer adoption barriers than a new language.

As software engineers, we can run case studies of projects that attempt to adopt such frameworks. By documenting how and why the adoptions succeed or fail, we can better understand the important context variables for successful framework reuse.

The scope of observations and conclusions we present here are limited to the populations with which we interacted. We spoke mainly to computational scientists in either academia or government agencies who use computers to simulate physical phenomena. There are many other applications for HPC (for example, signal processing, cryptography, and 3D rendering), and HPC is also used in industry (for example, movie special effects, automobile manufacturing, and oil production).

Scientific-software systems are growing larger and more complex. We're finally starting to see interaction among the computational-science and SE communities, but more dialogue and studies are necessary. We've much to learn from each other.

7 CONCLUSIONS

7.1 Technology Transfer

In early 2009 we were nearing the end of our 4 year contract. Over this past year we had several meeting with Dr. Jeffery Vetter of Oak Ridge National Laboratory (ORNL) in order to help him identify sites that could participate in future productivity studies. In addition we completed any software development and database work to make our tools more portable. Our final task was to transfer our tools and support software to ORNL so that Dr. Vetter can use the results of our work after our involvement ends. Toward this end we prepared a CD containing our software and wrote an installation and user guide for our software.

On April 22, 2009 we met with Dr. Vetter and helped him install the software on an appropriate Linux machine at Oak Ridge. The installation and user guide is reproduced as Appendix B of this report.

Over the course of this study, we achieved the following general results:

1. We developed a protocol applicable to measuring productivity in the parallel programming domain. We have tested this protocol over 25 times at 8 universities across the United States (Iowa State University, Massachusetts Institute of Technology, Mississippi State University, University of California San Diego, University of California Santa Barbara, University of Hawaii, University of Maryland, University of Southern California) (Section 2).
2. We helped to refine a measure of productivity that incorporates both development time and performance into a single productivity number (Section 3).
3. In order to aid other researchers studying similar environments, we developed an Experiment Manager tool for collecting data on the development of parallel programs, as well as a suite of tools to aid in the capture and analysis of such data (Section 4).
4. We studied several large scale development environments (such as the ASC Alliance centers) in order to better understand the environment used to build large parallel programming applications (Section 5). This included several surveys and interviews with many professional programmers in these environments (Section 6).

7.2 Lessons learned

As part of this study we learned much about the development and measurement of productivity in high performance computing environments. Many of these results simply confirm previous conclusions we have made in our over 25 years of experience with the NASA Goddard Space Flight Center's Software Engineering Laboratory (SEL) from 1976 through 2002 [2].

Some of the more important results are as follows:

1. ***It is possible to run multi-university studies to collect data usable to evaluate HPC productivity.*** We were able to achieve much of our original purpose for this grant. As described in this report, and in the associated 21 publications that this grant produced, we have run 25 studies across 8 universities to better understand how HPC programs are development. We were able to develop an experimental protocol that could be used at multiple locations for the execution, data collection, and evaluation of experimental results. But the process is not easy, as the following lessons learned illustrate.

2. ***The conducting of studies in environments outside of the researcher's own laboratory is a non-trivial exercise.*** You must have "buy-in" from management, whether this is the faculty at other universities or management at the various ASC-Alliance centers or National labs. While we did achieve cooperation and run experiments at 8 universities, there were also continuing issues with the execution of these experiments. From our SEL experience: *Lesson 8: Having a shared commitment [between researcher and project management] over research and development is vital for success. Lesson 11: Having upper management support is important for continued success.* We found all of these true in our HPCS work. Getting the various organizations to understand the importance of the HPCS project and our role in it was always a stumbling block. We found the various faculty and professional HPC technical staff always available at meetings and on the telephone, but that did not always translate into actions back at their home locations. They often did not see how the HPCS program would affect themselves in their everyday lives.

   One of our failures in the last year of this project was to run various studies at various national labs. Because of security issues, we could not be present in their facilities, and we could not collect the data and process it at the University of Maryland. Therefore, we needed to find individuals who could install, run, and oversee such experiments in their own locations. The development of our user guide for the installation and operation of our toolset (reproduced as Appendix B in this report) was our way of offloading the work to relevant staff at these labs. While there seemed to be interest, the stumbling block always seemed to be "Who will pay for it?" As our SEL study observed: *Lesson 13: It is difficult to make an engineering organization aware of the importance of software engineering to their mission.* We obviously did not make an appropriate impression on the management of these centers that proper software development was an important concern. Raw number of FLOPS was not the only concern for solving many of the complex problems needing solutions today.

3. ***External events always influence progress and results.*** In our case we had continuing issues with our non-University of Maryland partners understanding the importance of the IRB requirements in conducting studies of students or programmers. In almost all cases, we lost at least 6 months in setting up an experimental protocol at each new location due to the inexperience of the researcher there to fully understand the importance and difficulty of securing IRB approval in sufficient time to run the study. Because of this, we were able to run fewer studies than we had originally planned.

4. ***It is vital to any such program to understand the role of data collection and the importance of securing a baseline in order to be able to measure improvement.*** One goal for the HPCS program was to generate a 10-fold improvement in software productivity from the 2002 base year. However, in order to measure what the new petascale-class of machines are capable of, we had to know the baseline from 2002. As we learned in our SEL experience: *Lesson 1: Data collection requires a rigorous process and professional staff. Lesson 5: Establishing a baseline of an organization's products, processes, and goals is critical to any improvement program.* In our experience with the HPCS program, we were unable to find any site with any productivity data from 2002, so a true 10-fold improvement, when the new class of machines are placed online, will not be measurable. In one case, one National Lab did have access to their 2002 machine, so we planned to run an experiment to try and develop some baseline data from that site. But in the process of setting up that experiment, the organization did not resolve their IRB issues quickly enough (see lessons learned 3, above), and by the time the approval was received, that machine had already been decommissioned.

7.3 Student research

Many students have been supported by this research. One MS degree was partially supported: Rola Alameh, 2007; 3 PhD dissertations were partially supported: Lorin Hochstein, 2006, Taiga Nakamura, 2007, and Daniela Cruzes, 2007 and one student is currently working on a PhD dissertation: Nico Zazaworka.

7.4 Publications

The following publications were partially sponsored by this grant:

1. Zelkowitz M., V. Basili, S. Asgari, L. Hochstein, J. Hollingsworth, and T. Nakamura, "Productivity measures for high performance computers," *Computer Society International Symposium on Software Metrics*, Como, Italy, September, 2005.
2. Lorin Hochstein, *Development of an empirical approach to building domain-specific knowledge applied to high-end computing*, PhD dissertation, University of Maryland, Computer Science, August 2005.
3. Hochstein L., V. Basili, M. Zelkowitz, J. Hollingsworth and J. Carver, "Combining self-reported and automatic data to improve effort measurement," *Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, Portugal, September, 2005.
4. Hochstein L., J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, M. Zelkowitz, "HPC Programmer Productivity: A Case Study of Novice HPC Programmers," *Supercomputing 2005*, Seattle, WA, November 2005 .
5. Shull F., J. Carver, L. Hochstein, and V. Basili, "Empirical study design in the area of high performance computing (HPC)," *Computer Society International Symposium on Empirical Software Engineering*, Noosa Heads, Australia, November, 2005, 305-314.
6. Lorin Hochstein, Victor R. Basili, "An Empirical Study to Compare Two Parallel Programming Models," *18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '06).* Cambridge, MA, July 2006.

7. Taiga Nakamura, Lorin Hochstein, Victor R. Basili, "Identifying Domain-Specific Defect Classes Using Inspections and Change History", *Proceeding of 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE'06)*, Rio de Janeiro, Brazil, September, 2006.
8. Jeffrey C. Carver, Lorin Hochstein, Richard P. Kendall, Taiga Nakamura, Marvin V. Zelkowitz, Victor R. Basili and Douglass E. Post, "Observations about Software Development for High End Computing," *CTWatch* (**2**)4, November 2006.
9. Lorin Hochstein, Taiga Nakamura, Victor R. Basili, Sima Asgari, Marvin V. Zelkowitz, Jeffrey K. Hollingsworth, Forrest Shull, Jeffrey Carver, Martin Voelp, Nico Zazworka, Philip Johnson, "Experiments to Understand HPC Time to Development," *CTWatch* (**2**)4 November 2006.
10. Nicole Wolter, Michael O. McCracken, Allan Snavely, Lorin Hochstein, Taiga Nakamura and Victor Basili, "What's working in HPC: Investigating HPC User Behavior and Productivity," CTWatch (**2**)4A:9-17, November, 2006.
11. Andrew Funk, Victor Basili, Lorin Hochstein and Jeremy Kepner, Analysis of parallel software development using relative development time productivity metric, *CTWatch* (**2**)4A:46-51, November 2006.
12. Taiga Nakamura, "HPCBugBase: an experience base for HPC defects," *Supercomputing 2006*, Tampa, FL, October 2006.
13. Rola Alameh, Nico Zazworka, Jeffrey K. Hollingsworth, "Performance Measurement of Novice HPC Programmers' Code," *Workshop on Software Engineering of High Productivity Computers 2007*, Minneapolis, MN, May 2007.
14. Taiga Nakamura, *Recurring software defects in high end computing*, PhD dissertation, University of Maryland, Computer Science, May, 2007.
15. J. Carver, "Post-Workshop Report for the Third International Workshop on Software Engineering for High Performance Computing Applications (SE-HPC 07)," *SIGSOFT Software Eng. Notes*, vol. **32**, no. 5, 2007, 38–43.
16. Daniela Cruzes, *Secondary Analysis on Experimental Software Engineering Studies*, PhD dissertation, Salvador University (UNIFACS), Salvador, Brazil, 2007
17. Victor Basili and Marvin Zelkowitz, "Empirical studies to build a science of computer science," *Communications of the ACM* (**50**)11:33-37, November 2007.
18. Rola Alameh, *Investigating the effects of HPC novice programmer variations on code performance*, MS thesis, University of Maryland, Computer Science, December 2007.
19. Lorin Hochstein and Victor R. Basili, "The ASC-Alliance Projects: A case study of large-scale parallel scientific code development," *IEEE Computer*, March, 2008, 50-58.
20. Lorin Hochstein, Taiga Nakamura, Forrest Shull, Nico Zazworka, Victor R. Basili, Marvin V. Zelkowitz, "An Environment for Conducting Families of Software Engineering Experiments," *Advances in Computers* **74**, Elsevier, 2008, 175-200.
21. Basili V. R., D. Cruzes, J. C. Carver, L. M. Hochstein, J. K. Hollingsworth, M. V. Zelkowitz and F. Shull, "Understanding the High-Performance Computing Community: A Software Engineer's Perspective," *IEEE Software*, July 2008, 29-37.

8. REFERENCES

[1] Dongarra J., R. Graybill, W. Harrod, R. Lucas, E. Lusk, P. Luszczek, J. Mcmahon, A. Snavely, J. Vetter, K. Ineyelick, S. Alam, R. Campbell, L. Carrington, "DARPA's HPCS Program: History, Models, Tools, Languages," *Advances in Computers* **72**, Elsevier, (2008) 1-101.

[2] Basili V., F. McGarry, R. Pajerski, M. Zelkowitz, "Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Laboratory," *IEEE Computer Society and ACM International Conf. on Soft. Eng.,* Orlando FL, May 2002, 69-79.

[3] MPI Forum, MPI: *A message passing interface standard, version 2.2,*High Performance Computing Center, Stuttgart, 2009.

[4] Chapman B., G, Jost, and van der Pas R, *Using OpenMP*, MIT, 2008.

[5] Basili V. Basili, S. Green, "Software Process Evolution at the SEL," *IEEE Software* 11(**4**), (July 1994), 58-66.

[6] Basili V., "Evolving and Packaging Reading Technologies," *Journal of Systems and Software*, vol. **38**(1) July 1997, 3-12.

[7] Basili V., F. Shull, and F. Lanubile, "Building Knowledge through Families of Experiments," *IEEE Transactions on Software Engineering*, vol. **25**(4), July 1999, 456-473.

[8] Hochstein L., J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, M. Zelkowitz, "HPC Programmer Productivity: A Case Study of Novice HPC Programmers," *Supercomputing 2005*, Seattle, WA, November 2005.

[9] Carver J., L. Jaccheri, S. Morasca, F. Shull, 'Issues in using students in empirical studies in software engineering education," *International Symposium on Software Metrics*, Sydney, Australia, (2003), 239-249.

[10] Miller J., "Applying meta-analytical procedures to software engineering experiments," *Journal of Systems and Software* **54**(1), (September, 2000) 29-39.

[11] Budinsky F., S. A. Brodsky and E. Merks, *Eclipse Modeling Framework*, Pearson Education, 2003.

[12] Spacco J., J. Strecker, D. Hovemeyer, W. Pugh, "Software repository mining with Marmoset: an automated programming project snapshot and testing system," *Proceedings of the 2005 International Workshop on Mining Software Repositories*, St. Louis, Missouri, (2005), 1-5.

[13] Johnson P. M., H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa and T. Yamashita, "Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH," *Proceedings of the 2004 International Symposium on Empirical Software Engineering,* Los Angeles, California, August, 2004.

[14] Hochstein L., V. Basili, M. Zelkowitz, J. Hollingsworth and J. Carver, "Combining self-reported and automatic data to improve effort measurement," *Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005),* Lisbon, Portugal, September 2005, 356-365.

[15] Asgari S., L. Hochstein, V. Basili, M. Zelkowitz, J. Hollingsworth, J. Carver, and F. Shull, "Generating Testable Hypotheses from Tacit Knowledge for High Productivity Computing," *2nd International Workshop on Software Engineering for High Performance Computing System Applications,* St. Louis, MO, May 2005, 17-21.

[16] Post D., Kendall, R.P., and Whitney, E. "Case study of the Falcon Project," *Second International Workshop on Software Engineering for High Performance Computing Systems Applications,* St. Louis, MO, 2005.

[17] Butenhof D., *Programming with POSIX Threads*, Addison Wesley, 1997.

[18] Hochstein L., T. Nakamura, F. Shull, N. Zazaworka, V. Basili and M. Zelkowitz, "An Environment for Conducting Families of Software Engineering Experiments," *Advances in Computers* **74**, Elsevier, Boston MA (2008) 175-200.

[19] *The International Journal of High Performance Computing Applications,* (**18**)4, Winter 2004.

[20] Gard4ner M., "Mathematical games," *Scientific American,* October, 1970.

[21] Alameh R., *Investigating the effects of HPC novice programmer variations on code performance,* MS thesis, University of Maryland, Computer Science, December 2007.

[22] Keller, J., K. Christoph, and J. Träff , *Practical PRAM Programming*, John Wiley and Sons, 2001.

[23] Vishkin U., S. Dascal, E. Berkovich, and J. Nuzman, "Explicit Multi-Threading (XMT) Bridging Models for Instruction Parallelism," *10th ACM Symposium on Parallel Algorithms and Architectures (SPAA),* 1998.

[24] Hochstein L., V. R. Basili, "An Empirical Study to Compare Two Parallel Programming Models," *18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '06).* Cambridge, MA, July 2006.

[25] Nakamura T., L. Hochstein and V. R. Basili, “Identifying Domain-Specific Defect Classes: Using Inspections and Change History,” *International Symposium on Empirical Software Engineering, (ISESE)*, Rio de Janeiro, September, 2006.

[26] Zelkowitz M. , V. Basili, S. Asgari, L. Hochstein, J. Hollingsworth and T. Nakamura, “Measuring productivity on high performance computers,” *IEEE Symp. on Software Metrics*, Como, Italy, (September 2005).

[27] Nakamura T., *Recurring software defects in high end computing*, PhD dissertation, University of Maryland, Computer Science, May, 2007.

[28] Vesperman J., *Essential CVS*, O’Reilly Media, 2003.

[29] Cownie J. and S. Moore, Portable OpenMP debugging with Totalview, *Proceedings of the Second European Workshop on OpenMP (EWOMP'2000)*, 2000.

[30] Hastings R., B Joyce, “Purify: Fast detection of memory leaks and access errors,” *Proceedings of the Winter USENIX Conference*, 1992.

[31] Stallman R. and R. Pesch, “Debugging with GDB: the GNU source-level debugger,” Free Software Foundation, 1995.

[32] CMMI Team, *CMMI for Development, Version 1.2*, Carnegie Mellon University Software Engineering Institute, CMU/SEI-2006-TR-008, Software Engineering Institute, August 2006.

9. ACRONYMS

APMS	Automated Performance Measurement System
ASC	Accelerated Scientific Computing
ASCI	Accelerated Strategic Computing Initiative
CAD	Computer Aided Design
CCA	Common Component Architecture
CMMI	Capability Maturity Model Integration
CPU	Central Processing Units
CVS	Concurrent Version System
DARPA	Defense Advanced Research Projects Agency
DBD	Data Base Driver
DBI	Data Base Interface
DBMS	Data Base Management System
DOE	Department of Energy
DB	Database
FLOPS	Floating Point Operations per Second
GDB	GNU Project Debugger
GNU	GNU's not UNIX
GUI	Graphical User Interface
HEC	High End Computing (same as HPC in this report)
HPC	High Performance Computing
HPCS	[DARPA] High Productivity Computing System [Program]
IDE	Integrated Development Environments
IRB	Institutional Review Board
KLOC	Kilo (thousands of) Lines of Code
MPI	Message Passing Interface
NNSA	National Nuclear Security Administration
OO	Object-Oriented
OpenMP	Open Multi-Processing [interface]
POOMA	Parallel Object-Oriented Methods and Applications
POSIX	Portable Operating System Interface
PRAM	Parallel Random Access Machine
SE	Software Engineering
SEL	[NASA Goddard Space Flight Center] Software Engineering Laboratory
SMP	Symmetric Multiprocessors
UMD	University of Maryland
XMT-C	Explicit Multi-threading C

APPENDIX A: LIST OF HPC FOLKLORE

Initial List	Updated List
[1] Use of Parallel machines is not just for more CPU power, but also for more total memory or total cache (at a given level).	[1] Many people use parallel machines primarily for the large amount of memory available (cache or main).
[2] It's hard to create a parallel language that provides good performance across multiple platforms.	[2] It's hard to create a parallel language that provides good performance across multiple platforms.
[3] It's easier to get something working in using a shared memory model than message passing.	[3] It's easier to get something working using a shared memory model than message passing.
[4] It's harder to debug shared memory programs due to race conditions involving shared regions.	[4] Debugging race conditions in shared memory programs is harder than debugging race conditions in message passing programs.
[5] Explicit distributed memory programming results in programs that run faster since programmers are forced to think about data distribution (and thus locality) issues.	[5] Explicit distributed memory programming results in programs that run faster than shared memory programs since programmers are forced to think about data distribution (and thus locality) issues.
[6] In master/worker parallelism, the master soon becomes the bottleneck and thus systems with a single master will not scale.	[6] In master/worker parallelism, a system with a single master has limited scalability because the master becomes a bottleneck.
[7] Overlapping computation and communication can result in at most a 2x speedup in a program.	[7] In MPI programs, overlapping computation and communication (non-blocking) can result in at most a 2x speedup in a program.
[8] HPF's data distribution process is also useful for SMP systems since it makes programmers think about locality issues.	[8] For large-scale shared memory systems, you can achieve better performance using global arrays with explicit distribution operations than using Open MP.
[9] Parallelization is easy, Performance is hard. For example, identifying parallel tasks in a computation tends to be a lot easier than getting the data decomposition and load balancing right for efficiency and scalability.	[9] Identifying parallelism is hard, but achieving performance is easy.
[10] It's easy to write slow code on fast machines.	[10] It's easy to write slow code on fast machines. Generally, the first parallel implementation of a code is slower than its serial counterpart.
[11] Experts often start with incorrect programs that capture the core computations and data movements. They get these working at high performance first, and then they make the code functionally correct later.	[11] Sometimes, a good approach for developing parallel programs is to program for performance before programming for correctness.
	[12] It’s better to write a program with fewer large messages than many small messages.

APPENDIX B: TOOLS DEVELOPED FOR HPCS STUDIES

User Manuals, Installation Guides
21 April 2009

B.1. Introduction

For the classroom studies we conducted a whole collection of tools has been developed. They can be categorized in the following way Figure 7).

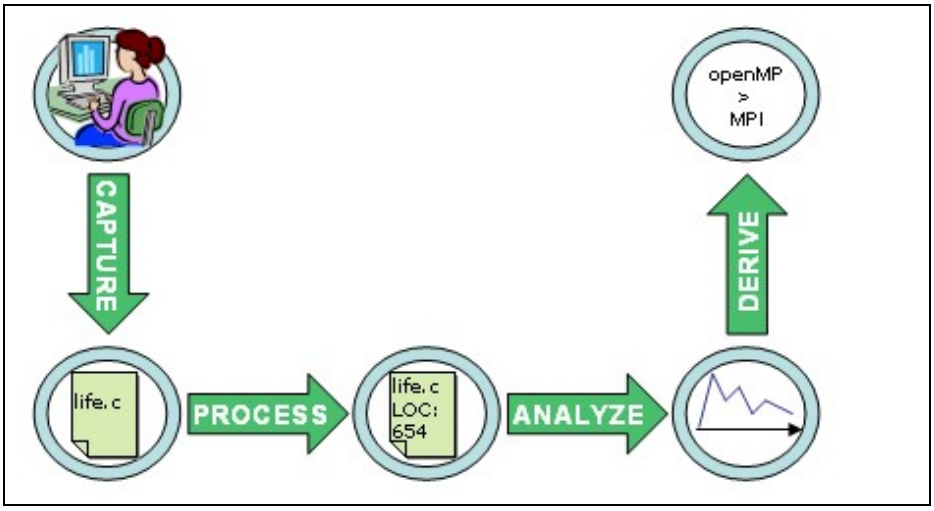

Figure 7: Research process and tool categories

- **capture tools**: help to gather data from study participants and join this data in our common data source – a relational DB
- **processing tools**: calculate / post process data in the DB to retrieve non captured and higher level data
- **analyze tools**: provide views on the DB in order to support the validation of hypotheses and to gain new insights
- **knowledge bases**: present the derived knowledge of analyze processes

Some of these tools have been developed especially for our research purposes and environments others are also applicable in similar studies or related research domains. For later ones we provide information about functionality, requirements and applicability. They can be downloaded or requested on the according tool sites.

Figure 8 shows the whole system framework and dataflow.

B.1.1 About this document

This document provides following information for each of the tools:

- **Purpose** of the tool and where it fits into the whole picture
- **Dependencies** of the tool that have to be installed prior to the tools installation
- **Further readings** about the functionality (mostly covered in student thesis)
- **Technical implementation** details, e.g. which languages and techniques were used to realize the tool.
- **Installation** routine to install the tool on a UNIX-like system.

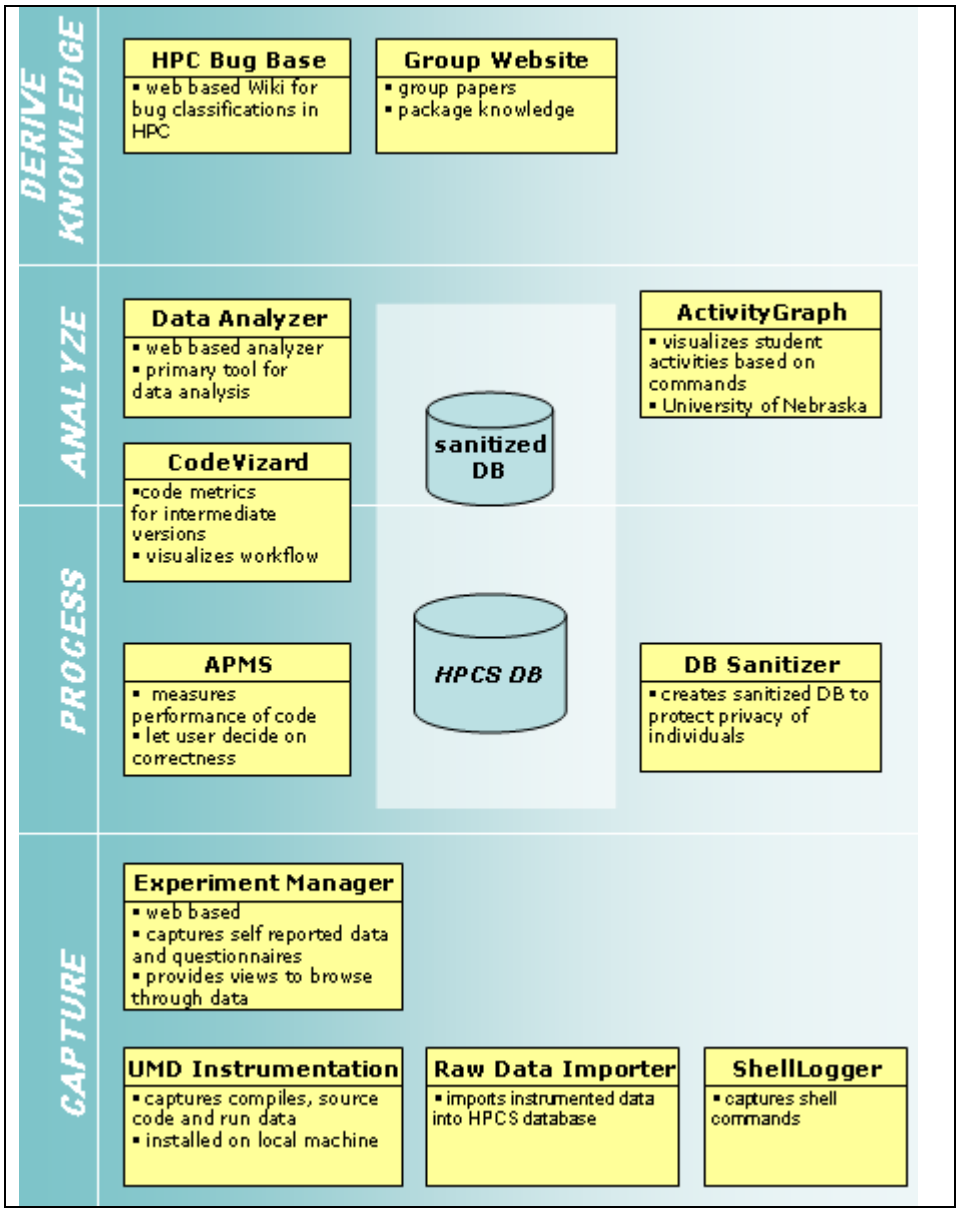

Figure 8: System framework

B.1.2 General Requirements

The UNIX environment should at least have following software packages installed:

- Apache web server that support execution of perl and php > 5 scripts and Postgresql perl module
- Python > 2.1
- Java > 1.5
- UNIX shell is used for interacting with the compilers. The software uses tcsh/csh or bash shell (latest version), but other shells may be acceptable.
- A graphical windows system (e.g., X-windows, MS Windows) for some of the analysis software
- Postgres > 8.1 (open source) Data Base Management System (DBMS)

B.1.3 More Information Sources

Can be found on the group website

- http://hpcs.cs.umd.edu

B.1.4 Contact Information

Prof. Victor R Basili: basili@cs.umd.edu
Prof. Marvin Zelkowitz: mvz@cs.umd.edu
Nico Zazworka (developer): nico@cs.umd.edu

B.1.5 Disclaimer

B.2 Database

B.2.1 Purpose

The database is the central point for data storage. Almost all applications store or read at some point data into or from the database.

B.2.2 Technical Implementation

We used postgresql as relational DBMS. The system needs one database (with one schema). The 67 tables are defined in a create SQL script. Some initial data has to be filled (e.g. login roles).

B.2.3 Dependencies

- Postgresql 8.3 or higher has to be installed
- Rights to create new databases and to restore backup files

B.2.4 Installation

- Create a new database with name hpcs
  CREATE DATABASE hpcs
  WITH OWNER = postgres
  ENCODING = 'UTF8';
- Use pgrestore the backup file on the installation CD to create the DB
  - pg_restore.exe -h localhost -p 5432 -U postgres -d hpcs -v "\software\Database\hpcs_schema.backup"
- This will create all necessary tables and one Experiment Manager user "**Administrator**" with password "**000000**"

B.3 Experiment Manager

B.3.1 Purpose

This web based manager provides functionality for different roles (roles are decided based upon login role):

- Study Participant
  - Maintain manually collected effort information
  - Maintain manually collected defect information
- Professor / Instructor
  - Create assignments (set due date, etc…)
- Administrator
  - Create classes / experiments
  - Maintain all meta information (e.g. programming languages, clusters,…)
  - Insight into all collected data / classes

B.3.2 Dependencies

- Database

B.3.3 Further Reading

Further information about the functionality provided can be found in:
/doc/ExperimentManager/ainc-exp-mgr-071107.doc

B.3.4 Technical Implementation

This web based application is implemented as a collection of perl scripts. Your web server has to be set up to execute these scripts.

B.3.5 Installation

- Copy the umdexpt directory from the installation CD to a directory of your choice (e.g. foobar)
    - Set up apache to run scripts in foobar/umdexpt/code/cgi-bin
        - Add a new script alias to your http.conf (typically stored in directory: etc/httpd/conf/httpd.conf)

ScriptAlias /umdexpt/cgi-bin/ „/foobar/code/cgi-bin/“

        - Add one alias to the code folder (needed for pictures)

Alias /umdexpt/ “/foobar/code/”

        - Enable cgi execution in http conf:
          AddHandler cgi-script .cgi
        - Eventually you have to restart apache:
          /usr/sbin/apachectl –k restart
    - Make sure the perl interpreter is reachable in /usr/local/bin/perl
        - If its not there create a symbolic link:
          ln myperl_directory /usr/local/bin/perl
- Further you might need to install the Postgres Data Base Driver (DBD::Pg) so that perl can communicate via the Data Base Interface (DBI) with the postgresql database:
    - yum install perl-DBD-Pg
- Now as last step the Configuration.pm in the cgi-bin folder has to set up properly
    - Set up the name of the database you installed earlier
    - Set up the port of the database
    - Set up email information
    - (optionally) set up the Hackystat server
- If you set up the database with the script described above you now can log in as “**Administrator**” with password “**000000**”

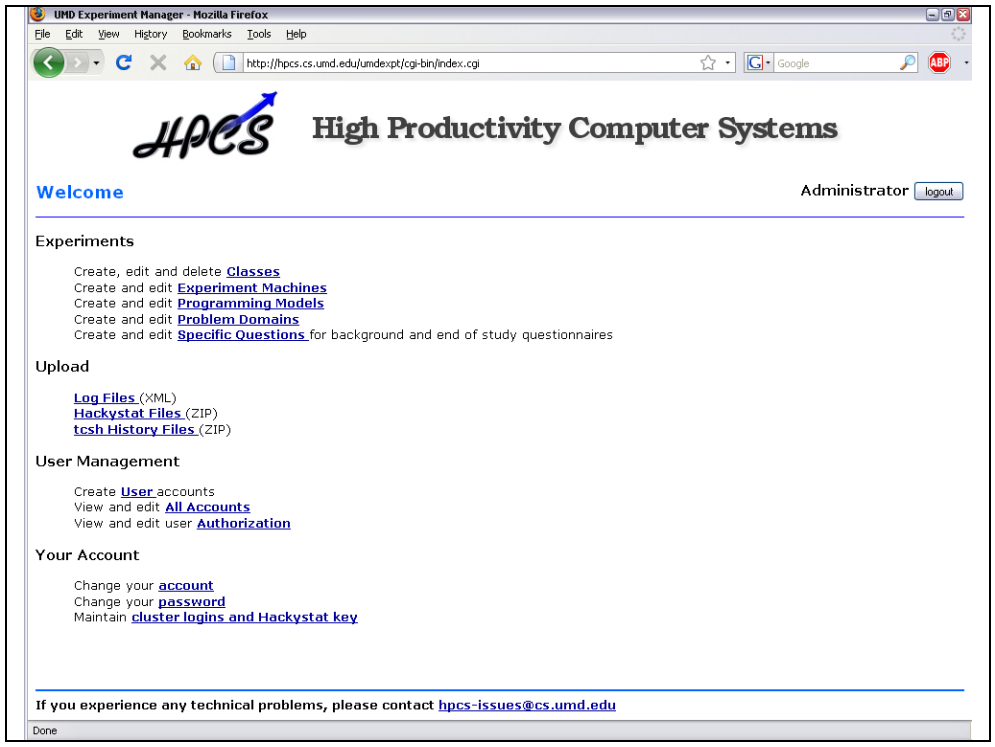

Figure 9: Administrator view of Experiment Manager

B.4 Raw Data Importer

B.4.1 Purpose

Parses the captured data from the instrumentation package (xml files) and saves it into the database.

B.4.2 Dependencies

- Database
- Data from UMD Instrumentation

B.4.3 Further Reading

- Master Thesis by Nabi Zamani:
  - */docs/RawDataImporter/ThesisNabiZamani.pdf*
- The tool was refactored based on a first implementation by student Andreas Anstock, his thesis can be found here:
  - */docs/RawDataImporter/ThesisAndreasAnstock.pdf*

B.4.4 Technical Implementation

The program is written in Java and uses Swing to create a graphical user interface. Therefore the tool should be started within a window system (e.g. MS Windows, XWindows, MacOS, or any other system that supports java GUI).

B.4.5 Installation

No installation needed. Simply run the Raw_Data_Importer.jar in *\software\RawDataImporter\RDI_26SEPT2007\RDI_26SEPT2007*

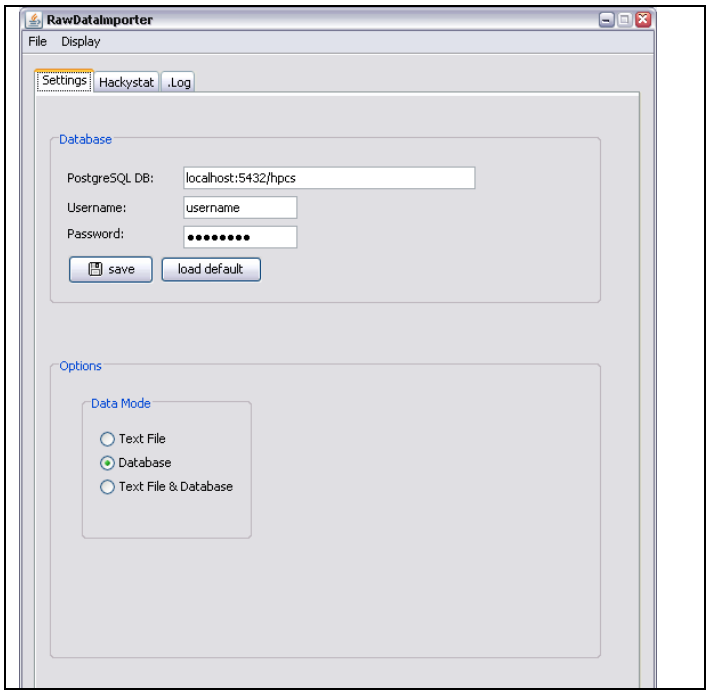

Figure 10: GUI interface of raw data importer

B.5 DataSanitizer

B.5.1 Purpose

This tool allows to copy “sanitized” data from one database to another. This was needed in our context to create databases without private information (that could identify students).
The tool enables the user to define a template (or privacy policy) that decides which data can be copied, which data is not copied, and which data is copied after applying transformations.

B.5.2 Dependencies

- The original hpcs database that serves as source
- A second database that serves as data sink
- Both databases have to be portgesql DBMS

B.5.3 Further Reading

The according thesis by Nico Zazworka can be found in:
/docs/DataSanitizer/ThesisNicoZazworka.pdf

B.5.4 Technical Implementation

The tool is implemented in java without a GUI. It can be called from the shell.

B.5.5 Installation

No installation needed, to run the program call:

java –cp .;postgresql-8.1-406.jdbc3.jar dataSanitizerController/Sanitizer

This will show you the available options:

Database Sanitizer Tool 0.1 dev
University Of Maryland

(4) to generate a privacy policy XML file for the source database:
Sanitizer g <sourcedbms: user@server:port> <databasename> <privacy policy XML file>
(2) to sanitize the data based on a description file in the target database:
Sanitizer s [options] <sourcedbms: user@server:port> <targetdbms: user@server:port> <databasename for targetdb> <privacy policy XML file>
options: -limit n sets a limit of sanitized rows for each table (for testing) –testset filename n creates a html file to observe n sanitized rows per table after the process
(3) to create HTML documentation files for admins and analysts based on a description file:
Sanitizer –doc [filename of the description file] [filename for adminDoc] [filename for analystDoc]

(4) to start the java swing gui
Sanitizer –gui

to show the help
Sanitizer –h

B.6 HPCBugBase

B.6.1 Purpose

The bug base is a knowledge base, realized as a web wiki. It contains information about occurring defects in high performance computing. The knowledge is derived knowledge from the studies.

B.6.2 Dependencies

None. All data in here is not directly linked to data, it is rather summarized knowledge.

B.6.3 Further Reading

/doc/HPCBugBase/DissertationTaigaNakamura.pdf

B.6.4 Technical Implementation

The wiki platform is media wiki (the same Wikipedia uses)

B.6.5 Installation

No installation is needed, the wiki is available at: http://hpcbugbase.org

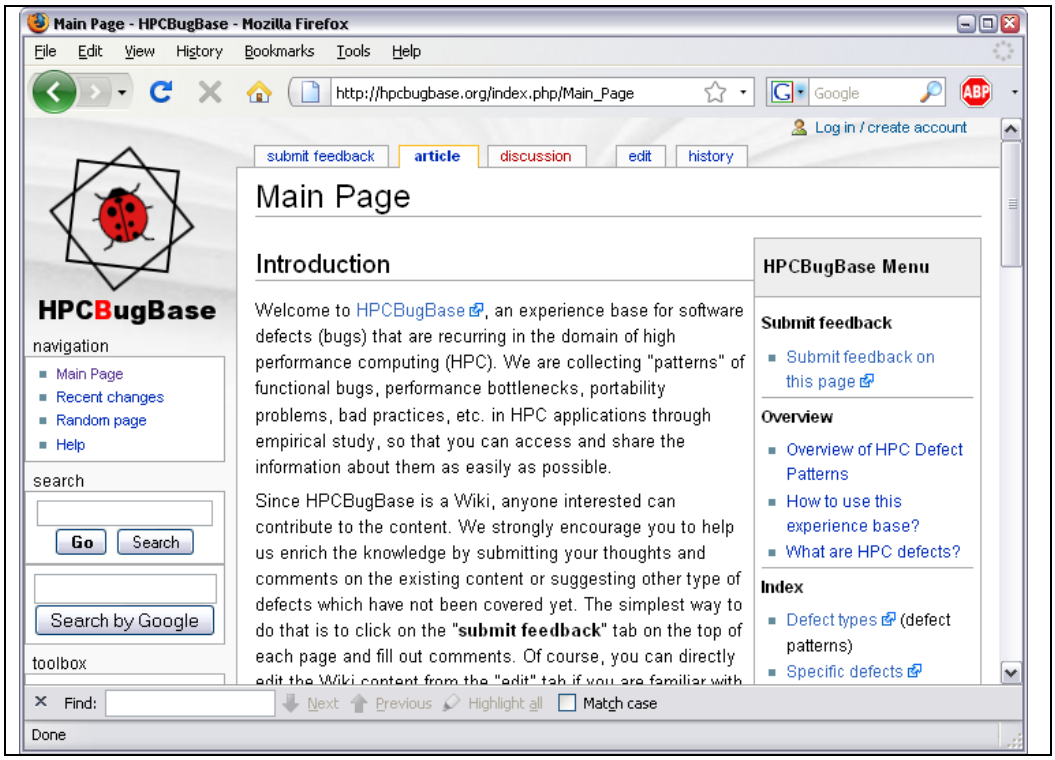

Figure 11: HPCBugBase

B.7 APMS - Automated Performance Measurement System

B.7.1 Purpose

In order to judge the performance of implementations students developed, we created a system that let the student codes run with various input parameters (e.g. size, iterations) and capture the performance data automatically.

B.7.2 Dependencies

- The system was tailored to one of our clusters, especially to its job submission procedure. Therefore it will very likely not run without modifications on another parallel cluster
- The system measured performance parameters through the PAPI API.

B.7.3 Further Reading

This work was Rola Alameh's Master thesis:
/doc/APMS/ThesisRolaAlameh.pdf

B.7.4 Technical Implementation

The system has a web based user interface that let the analyst submit source code and possible input variables. It was realized as java server pages (jsp).

B.7.5 Installation

Depending in the webserver configuration the files in */software/APMS/web/* have to be places where the web server can execute them as jsp files. The other directories are packages that the jsp pages access, they have to be placed in the web folders /lib folder (essentially in the folder the java virtual machines looks up packages – the classpath).

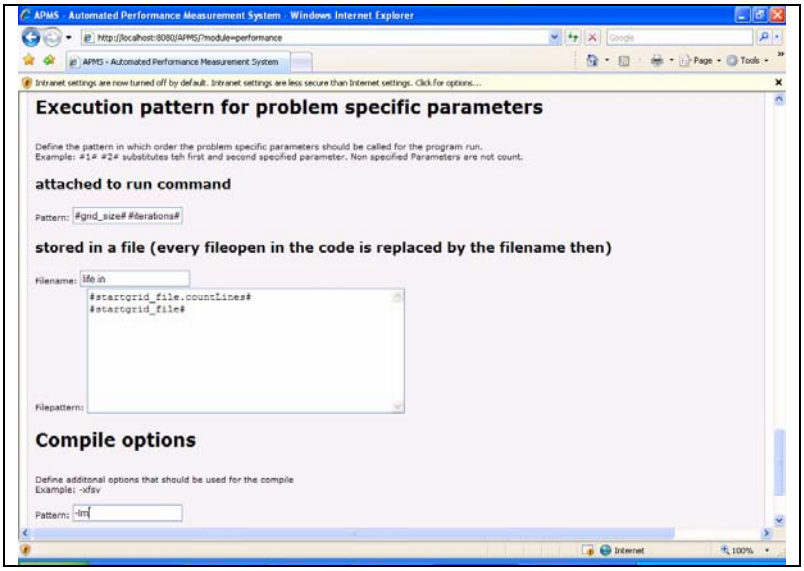

Figure 12: APMS

B.8 CodeVizard

B.8.1 Purpose

CodeVizard is a tool to visualize data captured from the studies. It can be used to identify common obstacles in computing as well as patterns.

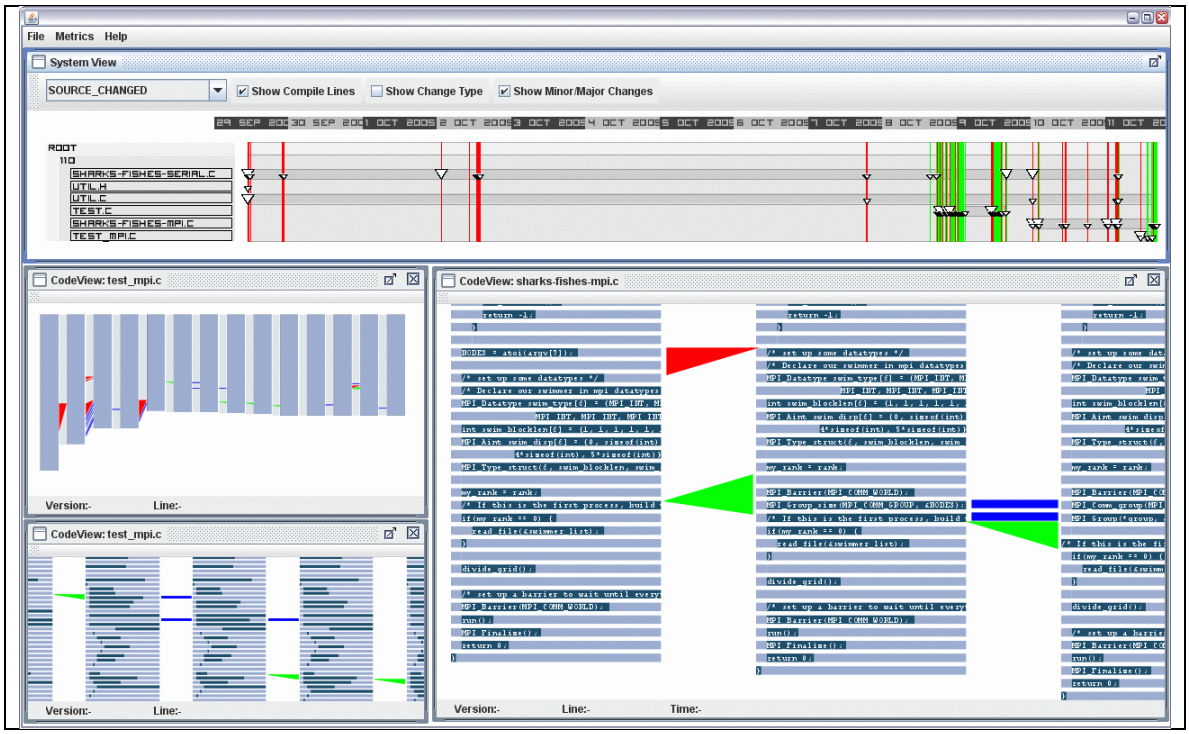

Figure 13: Code Vizard

Usage: Once the application is started click in the main menu on “Views” -> “Open Systemview”. This will bring up a new window – the System View. Here choose a visualization directory (usually: “/”), the number of folders (usually “all folders”) and the resolution (choose according to study length, typically in terms of days or weeks). Then click on the “go” button. Once the view is populated, you can drag with the **left** mouse button and zoom with the **right** mouse button (and dragging to the left (zoom out) and right (zoom in)).
Each triangle represents a file version that has been modified. By double-clicking with the left mouse button on a triangle the Code View is opened for that file. Notice that this might take some while (depending on the file size and number of versions – some minutes). The Code View lays out each file version and visual diffs between. The same dragging and zooming features as in the System View are available here.

Further details about coloring can be found in Figure 14.

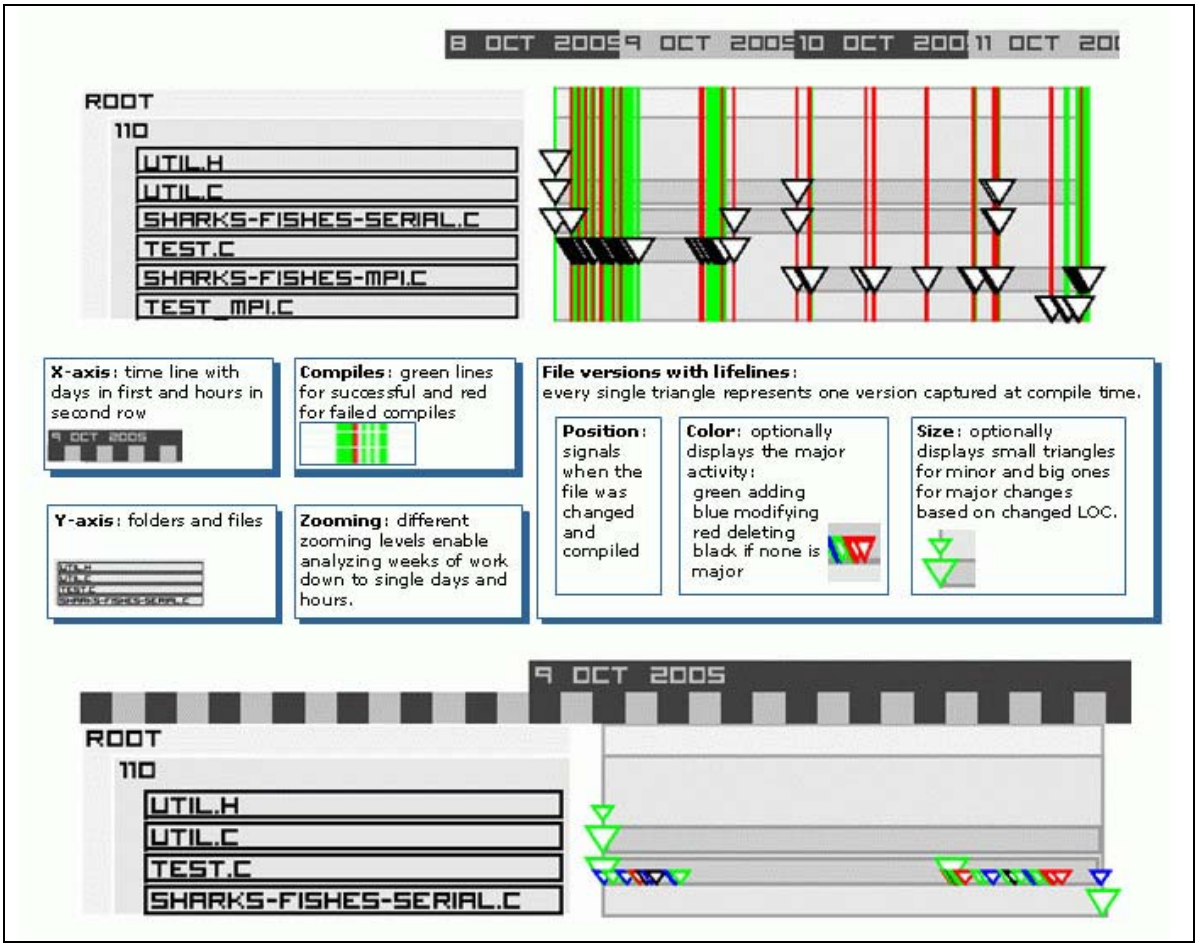

Figure 14: Code Vizard color codes

B.8.2 Dependencies

- Database with captured data in it

B.8.3 Further Reading

Thiago worked with the tool to study programmer behavior for NVIDIA Cuda/ GPU cards. His analysis provides examples how to use the tool:
/docs/CodeVizard/ThesisThiagoCraveiro.pdf

B.8.4 Technical Implementation

The tool is developed in Java with a GUI frontend. There are several views that show (1) the evolution of one source file (CodeView) or (2) the evolution of multiple files in a system (SystemView).

B.8.5 Installation

The tool does not need any installation. Copying the files from */software/CodeVizard/* to a directory is sufficient. It can be run with following parameters:

call pattern: [] means optional
DBREADER
db_url (format: jdbc:postgresql://localhost:5433/hpcs)
username
password
account_id (account_id(s) in the DB: a comma separated list: "66" or "66,89,108")
filename (one filename to show in the code view or "*wholeAccount*" to show the system view)
MAX_VERSIONS:int (max versions to be loaded, set to a very high number if all versions should be read)
VERSION_JUMPS:int (take only every nth version, set to 1 to read all versions)
ZOOM_FACTOR:float (horizontal time zoom factor for the system view, the lower the more zoomed in, 0.5 gives about 3 weeks on a screen)
Shelllogger-file:filename1 (name of the log file to be parsed - usually: name.hist -, needs to be placed in the CodeVizard folder, or “null” if no hist file present)
Shelllogger-file:filename2 (name of the log file to be parsed - usually: name.hist.2 -, needs to be placed in the CodeVizard folder, or “null” if no hist file present)
flatten directories: true/false (flattens all directories to one)

Examples:

To visualize the history of one single file of one subject (subject #32), all versions, and two history files:
java -cp ./bin;./lib/postgresql-8.1-406.jdbc3.jar;./lib/piccolox.jar -Xmx512m main.Main DBREADER jdbc:postgresql://localhost:5431/hpcs dbUser dbPassword 32 sharks-fishes-mpi.c 10000 1 1 0.5 student32.hist student32.hist2 true

To visualize the history of all files of one subject (subject #32), all versions, and no history files:
java -cp ./bin;./lib/postgresql-8.1-406.jdbc3.jar;./lib/piccolox.jar -Xmx512m main.Main DBREADER jdbc:postgresql://localhost:5431/hpcs dbUser dbPassword 32 *wholeAccount* 10000 1 1 0.5 null null true

To visualize the history of all files of one subject (subject #32), every 10^{th} version, and no history files:
java -cp ./bin;./lib/postgresql-8.1-406.jdbc3.jar;./lib/piccolox.jar -Xmx512m main.Main DBREADER jdbc:postgresql://localhost:5431/hpcs dbUser dbPassword 32 *wholeAccount* 10000 10 1 0.5 null null true

To visualize the history of all files of three subjects (subject #32,#64,#77), all versions, and no history files (this might take long to load and lot of memory depending on how much data is captured for each of the subjects):

java -cp ./bin;./lib/postgresql-8.1-406.jdbc3.jar;./lib/piccolox.jar -Xmx512m main.Main DBREADER jdbc:postgresql://localhost:5431/hpcs dbUser dbPassword 32,64,77 *wholeAccount* 10000 1 1 0.5 null null true

B.9 UMDInst

B.9.1 Purpose

The UMD Instrumentation package (UMDInst) is a collection of wrapper programs that collect data from study participants during their programming phases. Whenever subjects are compiling or running their codes snapshots of their programs and commands are written to a local log file. At the end of the study this log file contains all intermediate versions of the whole development process of HPC programs. Beside that it is possible to calculate additional measures based on this data, e.g. effort spent for an assignment.

B.9.2 Installation Guide

This document describes how to build and set up umdinst, an instrumentation package for collecting software development-related data from high-performance computers.

umdinst must be installed in one user's account (in a classroom study, this would typically be the TA's account). Then, it must be enabled at the beginning of the study. Once properly set up, umdinst works transparently in background so your work will not be disturbed.

For any queries about umdinst usage, please contact the hpcs group in the Experimental Software Engineering Group at the University of Maryland (hpcs-issues@cs.umd.edu).

B.9.2.1 How it works

There are three distinct types of data collection tools in umdinst: wrappers, Hackystat edit sensors, and Hackystat shell sensors. **Java 1.5 or higher is a requirement** in order to use all Hackystat sensors.

Wrappers are programs that replace existing programs used in development. Typically, only compilers are "wrapped". However, on systems that use batch scheduling, the job submission program is also wrapped. When invoked, the wrappers record data into a logfile, and then invoke the original program.

Hackystat edit sensors are plugins to editors such as Emacs or vi which collect data on files as they are being edited.

Hackystat shell sensors record shell commands. These sensors are from the Hackystat system.

B.9.2.2 Setup instructions

The setup instructions will install the main umdinst package. Each user to be instrumented will be required to run an additional script (see below).

- Make sure you know the identifier that is used in UMD Experiment Manager to refer to this class or study. You will need the study identifier to install the software.
- Extract files from the umdinst.tar.gz file on the HPC machine.
- Edit the setup.cfg file and modify the following variables

- mode: the instrumentation mode. Add a new line 'mode=ind'.
- id: the study identifier used by Experiment Manager
- javapath: the full path of the "java" program on the machine
- target_compilers: the location of the compilers to instrument.
- target_batches: the location of batch job submission programs, if the system uses batch-scheduling (e.g. PBS, LoadLever) to run jobs. For PBS, this program is usually called "qsub". If your system doesn't use batch-scheduling just specify nothing: 'batches ='.
- target_interactives: programs used to run a job interactively. Typically this is called "mpirun" for MPI programs.
- target_profiler: a program used for profiling an application (currently only one profiler may be specified)
- target_profiler_outputfile: the name of the file generated by the profiler. If specified, the data in this file will be captured. (Currently, only supports capturing this data if the file name can be specified in this file)
- target_profile_reporter: a program that takes a file generated by a profiler and generates a human-readable report. The instrumentation only supports such profiler reporters when they generate their output on standard out (as opposed to a text file).

- Run "python setup.py install" to generate the wrappers.

The setup program will create the following files and directories:

- umdinst/bin - directory which will contain the instrumented programs and a script that will configure a user's account for data collection
- umdinst/lib - directory that will contain the necessary library files
- umdinst/share - directory that will contain the runtime configuration files

B.9.2.3 Setting up a user account for instrumentation

To set up an account for instrumentation, a Hackystat key must be obtained from our Hackystat server. If you haven't done so, read the guide for participants for more information.

Once a Hackystat key has been obtained, the user must run the umdinst/bin/instrument script with the Hackystat key as an argument. For example, if the instrumentation is installed in /home/lorin/umdinst, and the key is 083ab13846g, then they would invoke the script as follows:

$ /abc/umdinst/bin/instrument 083ab13846g
Configuring your account for data collection
Hackystat host/key valid.
Success
Hackystat host/key valid.
The .EMACS_FILE_DIR path has been saved.
Success
Hackystat host/key valid.
The HACKYSTAT_VIM_SENSOR_DATA_FILE property has been saved.
Success
Downloading Emacs sensor, Version: 7.5.912...
.....
.....

Downloading CLI sensor, Version: 7.5.912...
The ENABLE_CLI_SENSOR property has been saved.
Success
Your .cshrc file has been updated.
Your environment has been set up.
Please log out and log back in for modifications to take effect.
(You may see an error message in your initial logout, this is normal)

This will modify the PATH variable in their .cshrc file, as well as add some Hackystat-related files to their directory. The user will need to log out and log back in for the instrumentation to take effect.

All data will be stored in the ~/.umdinst directory under the user's home directory.

B.9.2.4 (Optional) capturing program executions

By default, the instrumentation can only capture program executions if a wrapped program is involved in executing them (e.g. mpirun, qsub). However, it is often of interest to capture a run of a standalone executable program. To handle this, there is an additional program called run in umdinst/bin. This program can be used to execute another program, and capture information. For example:

$ gcc -o myprog myprog.c
$ run ./myprog

B.9.2.5 Uninstalling the instrumentation

Currently, there is no automatic way to disable the instrumentation. To do so manually requires the following steps:

- Delete the ~/.umdinst directory. **Important: make sure you backed up the content of umdinst/.data. Otherwise, all data will be lost!**
- Delete the umdinst directory where you installed the package
- Remove the residuals of the instrumentation environment.
    - Editing the ~/.cshrc file to remove the PATH modification to umdinst/bin
    - Editing the ~.emacs file
    - Removing the ~/.vim/plugin/VIMsensor.vim file (or the entire ~/.vim directory)
    - Removing the ~/.logout file
    - Removing the contents of the ~/.hackystat directory.

B.9.2.6 Setup Configurations

Install configuration options:

- Instrumentation mode: no default is set (prof, ind, none, or full)
    - **prof**: Save collected data in instrumentation account only
    - **ind**: Save collected data in participant's account only
    - **full**: Store collected data in both
    - **none**: Do not collect data

  Example: mode=prof (remove comment # in front)

- Identifier used to refer to this class/study: no default is set
  Example: id=class714 (remove comment # in front)

- Java executable path (must be Java 1.4 or greater), necessary unless you have disabled the Hackystat
  Example: featurejavapath=/usr/java/jdk1.5.0_11/bin/java

- Compilers to instrument (a colon-separated list)
  Example: target_compilers=/usr/bin/gcc:/usr/bin/mpicc

- Batch queue submision programs to instrument (a colon-separated list)
  Example: target_batches=/usr/bin/qsub

- Interactive job submission programs to instrument (a colon-separated list)
  Example: target_interactives=/usr/bin/mpirun

- *Debuggers to instrument. Can be either CLI (e.g gdb) or GUI (e.g. ddd), as a colon-separated list*
  Example: target_debuggers=/usr/bin/gdb

- Profiler for performance measurement. Currently, the instrumentation can handle only one profiler. In addition, the exact name of the profile output file must be known
  Example: target_profiler=
  Example: profiler_outputfile=

- Profile reporter that processes raw profile data and generate a report. The instrumentation only works if the reporter produces data on standard output
  Example: target_profile_reporter=/usr/bin/gprof

- Hackystat host (usually you don't have to change the default value). Necessary unless you have disabled the Hackystat feature. The default value is set to the UMD Hackystat server
  Example: hackystat_host=http://care.cs.umd.edu:8080/

- Additional path entries to be added
  Example: additional_path=

Default options that should not be changed:

- Installation directory (usually you don't have to change the default value). The default value makes the package compiled 'in-place' in the same directory the package source is placed.
  Example: prefix=.

- File that contains logins and Hackystat keys for users to be instrumented, if you know users' Hackystat keys in advance. Each line should contain a login, a tab (\t), and then a key. This must be readable and executable by all users who will be instrumented.
  Example: keyfile=keys.txt

- File that contains the logins of users that should be asked whether they are debugging
  Example: whitelistfile=ask.txt

- Encoding of the log file contents. base64 is safer for binaries, but quopri (quoted-printable) makes the file more human-readable, and smaller for most cases (base64, quopri or raw)
  Example: encoding=base64

B.9.2.7 Export data XML file schema

Note that the files exported do not have an opening XML declaration.

<log>: The file's root node; has no attributes

- **<compile>:** A <compile> element is recorded on each compiler invocation from the command line and is followed by zero or more <sourcefile> and <headerfile> elements containing the same number and kind of sub-elements, also most information duplicates with the compile elements (like the timestamps, they are just implemented this way for a historical reason); it has the attribute *success* that has value "1" if the file does not contain syxtax errors, else "0". The sub-elements of a compile are described below:
  - **<time>:** Linux timestamp of compile in number of seconds since midnight of the 1st of January 1970 UTC (Epoch); no attributes

    - **<timestr>:** Same timestamp in human readable format (WWW MMM DD HH24:MM:SS YYYY); no attributes
    - **<subject>:** Unique identifier for the participant; no attributes
    - **<command>:** Captured command entered from the subject in the system shell, contains global path of the called compiler, the source file name(s) and other compiler parameters; no attributes
    - **<time_interval>:** Used to be the compile time in seconds but is not used anymore (it now corresponds to the compile's *success* value); the actual compile time is found in the <time_interval> sub-elements of the <sourcefile> and <headerfile> elements; no attributes
- **<sourcefile>:** Contains source file of the previous compile tag; has an *success* attribute with the same value as the previous compile tag
    - **<name>:** Name of the sourcefile; no attributes
    - **<time>:** Same linux timestamp as in compile in number of seconds since midnight of the 1st of January 1970 UTC (Epoch); no attributes
    - **<timestr>:** Same compile timestamp in human readable format (WWW MMM DD HH24:MM:SS YYYY); no attributes
    - **<subject>:** Same unique participant identifier as in compile; no attributes
    - **<command>:** Same captured command as in compile, contains global path of the called compiler, the source file name(s) and other compiler parameters; no attributes
    - **<time_interval>:** Actual compile time in seconds; no attributes
    - **<path>:** Directory of the source file; no attributes
    - **<source>:** File content is encoded in the format specified in the *encode* attribute (default value: "base64"); no attributes
- **<job>:** Data captured when subject tried to run the last successfully compiled source code state; has attributes *type* with possible values "interactive" when mpirun is used, "batch" when a scheduler batch is used and "profiled" when a profiler is used.
    - **<subject>:** Same unique participant identifier as in compile; no attributes
    - **<time>:** Linux timestamp of the run in number of seconds since midnight of the 1st of January 1970 UTC (Epoch); no attributes
    - **<timestr>:** Same timestamp in human readable format (WWW MMM DD HH24:MM:SS YYYY); no attributes
    - **<command>:** Captured command for the execution of the program, contains global path of the called executable or batch file and other program or job batch file (scheduler) parameters; no attributes
    - **<path>:** Directory of the source file; no attributes
    - **<time_interval>:** Actual run time in seconds; no attributes

  In the case of type "batch" there is an extra element **<script>** containing the batch script with an attribute *encode* with the encode type.

  In the case of type "profiled" there is an extra element **<profiledata>** containing data about the profiler which cannot be Base 64 encoded.

- **<profile_report>:** This element and its sub-elements contain more detailed information about the report generated by the profiler
    - **<subject>:** Same unique participant identifier as in compile or job; no attributes
    - **<time>:** Linux timestamp of the call in number of seconds since midnight of the 1st of January 1970 UTC (Epoch); no attributes
    - **<timestr>:** Same timestamp in human readable format (WWW MMM DD HH24:MM:SS YYYY); no attributes
    - **<command>:** Captured command for running the profiler; no attributes
    - **<path>:** Directory in which the call occured; no attributes
    - **<time_interval>:** Actual run time in seconds; no attributes
    - **<contents>:** The actual report contents

- **<debug>:** This element and its sub-elements contain detailed information about the debugger called in the run; attribute *success* specifies if the run was successful (value "1") or if errors were found ("0")
    - **<subject>:** Same unique participant identifier as in other subject tags; no attributes
    - **<time>:** Linux timestamp of the call in number of seconds since midnight of the 1st of January 1970 UTC (Epoch); no attributes
    - **<timestr>:** Same timestamp in human readable format (WWW MMM DD HH24:MM:SS YYYY); no attributes
    - **<command>:** Captured command for running the debugger; no attributes
    - **<path>:** Directory in which the call occured; no attributes
    - **<time_interval>:** Actual run time in seconds; no attributes

B.9.2.8 Example file (1 successful compile + 1 unsuccessful run):

```
<log>

  <compile success=”1”>

    <time>1159378054</time>
    <timestr>Wed Sep 27 13:27:34 2006</timestr>
    <subject>student01</subject>
    <command>
      /clusterhomes/experimenter/umdinst/bin/mpicc MpiLife.c –o MpiLife –lm
    </command>
    <time_interval>0</time_interval>

  </compile>

  <sourcefile success=”1”>

    <name>MpiLife.c</name>
    <time>1159378054</time>
    <timestr>Wed Sep 27 13:27:34 2006</timestr>
```

```
  <subject>student01</subject>
  <command>
   /clusterhomes/experimenter/umdinst/bin/mpicc MpiLife.c –o MpiLife –lm
  </command>
  <time_interval>0.27</time_interval>
  <path>/clusterhomes/student01/life</path>
  <source encode=”base64”>
   […  base64 encoded file…]
  </source>

 </sourcefile>

 <job type=”interactive” success=”0”>

  <subject>student01</subject>
  <time>1159379274</time>
  <timestr>Wed Sep 27 13:47:54 2006</timestr>
  <command>
   /clusterhomes/experimenter/umdinst/bin/mpirun –np 2 ./MpiLife life.data.1.txt 5 60 60
  </command>
  <path>/clusterhomes/student01/life</path>
  <time_interval>0.63</time_interval>

 </job>

</log>
```

B.10 Shellogger

The tools allows to log every command that is entered through the Unix shell. It was never used in classroom and can is in experimental state.

The developer (Lorin) published it on google code:

http://code.google.com/p/shelllogger/

B.11 Hackystat

We used the Hackystat toolset (developed by Philip Johnson at the University of Hawaii) to capture further data from editors as vi and emacs. The visualization tools make at this moment no use of the data. However it can be stored in the database with the RawDataImporter.

The complete documentation how to set up an own hackystat server (or to use the one installed at the University of Hawaii) can be found on their google code website:

http://code.google.com/p/hackystat/

B.12 Guides

The next section contain guides that we developed for study participants and faculty/technicians.

B.12.1 HPCS Guide for Students / Study Subjects

Instructions for participants in HPC studies.

This is a guide for participants in HPC productivity studies. These instructions apply to participants of individual studies or of classroom studies, unless otherwise noted. When different instructions apply, the instructions will be marked appropriately:

- **(C)** applies to participants in **c**lassroom studies
- **(I)** applies to participants of **i**ndividual studies

Content:

1. At the beginning of the study
   1.a Create an account on Experiment Manager
   1.b Configure your account for instrumentation
   1.c Register your machine account names in Experiment Manager
   1.d Fill in background questionnaire
2. During the study
   2.a Logging your effort
   2.b Logging your defects
3. At the end of the study
   3.a Upload final submission
   3.b Fill in end-of-study questionnaire

At the beginning of the study

Create an account on Experiment Manager

Go to our Experiment Manager and create an account. Follow the link "If you are a student please enroll here!". Please make sure to have the Study ID (Class ID) for your study before registering:

- **(C)** If you are part of a classroom study, your professor should provide you with this information.
- **(I)** If you are part of an individual study please contact hpcs-issues@cs.umd.edu to get the appropriate Study ID.

After you register, you will receive an e-mail with a password to access the Experiment Manager system. Also included in this e-mail is a "Hackystat key", which you will need to configure your account in the next step.

Configure your account for instrumentation

The next step is to configure your account for instrumentation.

- **(C)** If you are part of a classroom study, please follow the instructions in this document.
- **(I)** If you are part of an individual study, please follow the instructions in this document. The complete instrumentation package can be downloaded from the umdinst page.

Register your machine account names in Experiment Manager

After you receive your Experiment Manager's account password in your e-mail, log in to the Experiment Manager. From the main page, follow the link "Maintain cluster logins and Hackystat key". Enter the login account(s) for the machines you will use. Click "save".

Fill in background questionnaire

To fill in the background questionnaire, from the main page of Experiment Manager, follow the link "Complete Background Questionnaire". Answer the questions in the form and click on "send" when you are finished. You will see a confirmation page if your form was successfully submitted.

During the study

Logging your effort (when you were instructed to do so)

If you have been asked in your study, use the experiment manager to keep track of how much time you spend working on the different assignments. Use the option "Report on your effort" from the main page of the Experiment Manager to do this.

You can enter the data manually indicating the details of your activities in the appropriate fields or you can use the "online Report Tool". The online tool is the easiest way to report your effort.

- To enter the data manually simply enter the start and stop time, and select the appropriate options for the other fields.
- The link to the "online Report Tool" can be found at the bottom of the Effort Page, where it says "Enter new data manually or use the Online Report Tool." The Online Report Tool allows you to track your time as your work, you can select different activities and record breaks by using the "Start/Break" button. Simply keep the small window on your desktop. Every time you start a new action, simply pull down the menu and switch activity. If you take a break or switch to another unrelated activity, simply click “break.” Remember to click “start” when you return.

Logging your defects (when you were instructed to do so)

If you have been asked in your study, use the experiment manager to keep track of any bugs that you encounter in your code (that take you more than 5 minutes to fix). Use the option "Report Defects" from the main page of the Experiment Manager to do this. Please provide information that is accurate and detailed as possible.

At the end of the study

Upload final submission

If you have been asked in your study, use Experiment Manager to upload your final submission for the assignment. Use the option "Upload final submission for an assignment" from the main page of the Experiment Manager to do this.

End-of-study questionnaire

Log in to the experiment manager and fill out the end of study questionnaire. This option will be available in the Experiment Manager, after your study was finished. The option is available from the main page in Experiment Manager under "Complete end of study questionnaire". Answer the questions in the form and click on "send" when you are finished. You will see a confirmation page if your form was successfully submitted.

B.12.2 HPCS Guide for Faculty / Technicians

The purpose of this document is to present the steps that should be followed to set up an in-class experiment. Links to examples and templates are included with each step.

Each of the steps of the checklist will be of one of the following types and will be tagged appropriately:

- **(Do once!):** indicates a step that should be performed only once for the whole semester for one class.

- **(Prepare once; do for each assignment!):** indicates a step that should be prepared once, but should be performed once for each assignment.
- **(Do for each assignment!):** indicates a step that should be performed for each assignment.

Some of the steps will be tagged with the text **(Technician)**. These indicate steps that we suggest to be performed by a technician assisting the professor, for example, a TA or system administrator.

Content:

1. Before class starts
   1.a Complete Human Subjects Forms and submit to your University Review Board (IRB) for approval
   1.b Register experiment machine and create class in Experiment Manager
   1.c Specify assignments for the students
   1.d Set up any additional artifacts for assignments
2. As soon as class starts
   2.a Ensure that the data collection mechanisms are set up
   2.b Have students sign consent forms and register in the HPCS repository
   2.c Have the students complete the Background Questionnaire
3. When the assignments begin
   3.a Explain the data to be collected and give out the homework assignment(s)
4. As soon as all assignments are done
   4.a Enable students to fill in the End-of-study Questionnaire
   4.b Make sure all experimental data and questionnaires are sent to UMD

Before the class starts

Complete Human Subjects Forms and submit to your University Review Board (IRB) for approval (Do once!)

Federal regulations require that all experiments, regardless of the type, that involve human subjects require such approval. This process is neither complicated nor difficult, but depending upon the university, this may require several months to accomplish. Start this early if this is your first time through the IRB process. Generally the IRB requires the experimenter to fill in forms that explain the goals and reasons for running the experiment, describe the tasks that are to be done by the subjects, and predict any harm that could come to the subjects as a result of participating in the study. We have had no trouble getting these forms approved in previous studies. An example of a completed form, based on the requirements at the University of Maryland, can be found here along with the attached answers to the required questions on the form. The consent form (see step 6) should also be approved by the University. An example of a consent form can be found here. Contact us for help if you are novice at this process.

Register experiment machine and create class in Experiment Manager (Do once!)

The goal of this step is to create a record to help you as well as the data collection and analysis team keep track of the relevant factors in the experiment.

Access the Experiment Manager system.

- If you are **not** registered in the system yet, you will need to contact the Administrator of the system and ask for an account. There is a link in the main page of the Experiment Manager to send an e-mail requesting an account. Once an account is created you will receive a user id and password.
- Log in using the account information that you received by e-mail.

Create an experiment machine.

- You need to register the machine (HPC hardware) that will be used in the experiment. To do so, use the option "Create experiment machine" from the main page of Experiment Manager. The screen will show a list of machines already registered.
    - If you have used the machine in previous experiments, you should see it in the list. Click on "HOME" to go back to main page.
    - If you don't see the machine in the list. Scroll down and fill in the form at the bottom of the page to register the machine. When you are finished click on "save".

Create a class for the experiment.

- From the main page of Experiment Manager, select the option "create and edit classes". You will see the experiments page.
- Click on "Create new experiment".
    - If you haven't registered the class previously, enter the information about the class. This information can be modified later, except for the class identifier that will be used by the students to sign up for the system. When you finish entering all information click on "save".

Make sure you inform the students of the class identifier. They will need it to register in Experiment Manager.

Specify assignments for the students (Do for each assignment!)

In this step an assignment specification needs to be created for each of the different combinations of computational models and problems.

Register assignment in Experiment Manager

- First, register your assignment with the HPCS data collection team in the Experiment Manager. From the main page, select "Create and edit classes". You will see a list of one or more classes previously registered. Click on the icon for Assignments in the row of class you want to create an assignment for. This will take you to the Assignments Page.
- To create a new assignment, click on "Create New Assignments". Fill in the required information for the assignment. This information can be edited, so most of this information can be filled in later if it is not available yet. When finished entering all information, click on "save".

Create assignment description to be given out to the students

You need to create an assignment description that can be given out to the students. By reusing descriptions from previous classes and staying within a template, you can help us ensure that student performance on the same problem will be comparable across classes.

Below are a series of assignment specification templates with example problem descriptions that you can reuse. In those documents, text in black represents the most common way of describing the problem – these should be kept “as is” to allow the comparison of results between classes. If you feel that a change is necessary, please let the Maryland team know at hpcs-issues@cs.umd.edu. Fields in green should be filled out according to the local context. Text in brackets provides hints as to what type of information should be included.

- Buffon-Laplace
- Game of Life
- Game of Life 2D
- Matrix Multiplication
- Parallel Sum
- Sharks & Fishes
- One file with all assignment specification templates

Select which students will do each assignment

You need to select which students will do each assignment depending on the experimental design chosen for your experiment. For more details on experimental designs please see these slides: Experimental Designs. Please contact us at hpcs-issues@cs.umd.edu if you need any help designing your experiment.

Once you decided on the experimental designs and created the assignments for the class, you need to indicate in Experiment Manager which students will do each assignment. From the main page, select "Create and edit classes". You will see a list of one or more classes previously registered. Click on the icon for Assignments in the row of class you want to give assignments to students. This will take you to the Assignments Page. For each assignment, click on the icon in the column "Students" and make sure that only the students that will be doing that assignment are selected. By default, all students will be checked. Uncheck the students who should nott be doing that assignment. When you are finished click on "save".

***(Optional)* Set up any additional artifacts for assignments (Do for each assignment!)**

In this step we suggest you create artifacts that could be handed out to the students together with the assignment specification in order to clarify the assignment and/or to ease the grading of the assignments. Examples of artifacts that could be handled out are test cases, examples of input files, etc.

Below is a list of examples of artifacts used in classes where the experiments were conducted that can be reused. We are starting to create this list, so if you have artifacts that you think could be useful for other faculty conducting experiments and would like to make them available from this page, please send an e-mail to hpcs-issues@cs.umd.edu.

Assignments and artifacts:

- **Dense matrix and vector multiplication**: Problem[1], Harness[1], Sequential Matlab code[1]
- **Game of Life**: Problem[1], Harness[1], Sequential Matlab code[1]
- **Sparse matrix and dense vector multiplication:** Problem[1], Harness[1], Sequential Matlab code[1]

[1] UCSB - CS240A: Applied Parallel Computing - Winter 2006 - Prof. John R. Gilbert; Assistant Viral Shah (http://www.cs.ucsb.edu/~cs240a/)

As soon as class starts

(Technician) Ensure that the data collection mechanisms are set up (Prepare once; do for each assignment!)

Ensure that you have the appropriate authorizations to modify the operating environment of the machines to be used and to send the student data to the University of Maryland. As with the IRB process, the process is not complicated, but does require planning. Do this **BEFORE** the class begins to avoid problems later.

It is necessary to ensure that these automated data collection mechanisms are functioning properly and that all data collection forms to be filled by the students are specified prior to commencement of the experiment. This step should be performed as soon as the instructor knows the accounts that will be used for the class.

Go to the download page to get the instrumentation package that contains all the necessary files. The instructions on how to install and run the instrumentation is found here. The instructor or the TA should install the instrumentation package. Each subject will need to run a script to make sure their accounts are configured. Before they start implementing a solution to their problems, they should notify the experimenter that the instrumentation package is properly configured. The experimenter should check if the data is being collected and notify the subjects to proceed with the implementation once that is verified before each assignment.

Please make sure to discuss with the system administrator of your organization of the experiment and the types of resources that the students will be using during the semester, to prevent technical problems and issues.

Have students sign consent forms and register in the HPCS repository. (Do once!)

In the step, the experimenter should ask the students to sign the consent forms. An example of a consent form can be found here.

Once the forms are signed, the experimenter should instruct the students participating in the experiment to enroll in the Experiment Manager. Instructions on hor to enroll in Experiment Manager are provided in the HPCS Guide for Students. They will need the class ID registered in the repository, so make sure that that have that information before they register.

Have the students complete the "Background Questionnaire" (Do once!)

Prior to beginning work on any of the assignments related to the experiment, each subject should log in into Experiment Manager and complete the Background Questionnaire. This form provides the researchers with important information about the background and experiences of the subjects in a variety of tasks that are related to software development for high performance computers. The data provided by the subjects will be used to analyze their final results and look for relationships between types of experience and outcomes of the experiment.

Before you hand out any assignments, make sure every subject has completed their background questionnaire.

When the assignments begin

Explain the data to be collected and give out the homework assignment(s) (Prepare once; do for each assignment!)

Make sure that the assignment description is complete and distribute it to the students.

In addition to any explanation of the programming models and the problems to be implemented, the instructors also need to explain to the subjects the effort and defect data that are being collected during this study. It is important that the subjects understand how to record their effort. The experimenters must be careful at this point not to inform the subjects of the reasons for collecting the data as this information may bias their working habits. The students should use the reporting options in Experiment Manager .

As soon as all assignments are done

Enable students to fill in the "End-of-study Questionnaire" (Do once!)

At the completion of all of the assignments that are part of the study, the subjects will each complete the "Post-Experiment Questionnaire." This questionnaire consists of a series of questions to help the researchers better understand what actually occurred during the study. The responses to these questions will also help the researchers more accurately analyze the data and interpret the results. The questionnaire is available in Experiment Manager, but it will only be accessible by the students when the professor enables it. To enable the questionnaire for the students:

- Log into Experiment Manager
- From the main page chose the option "Create and edit classes"
- Click on the red icon for "end of study". The icon changes to indicate that this feature has been enabled.
- The students will see a new option in their menu called "Complete end of study questionnaire".

Make sure all experimental data and questionnaires are sent to UMD (Do once!)

(Technician) E-mail hpcs-issues@cs.umd.edu all the log files from the instrumentation. Alternatively you can use the upload scripts function in Experiment Manager to send us the data.